proceedings

7th USENIX
Symposium on
Operating Systems
Design and
Implementation

Seattle, WA, USA November 6–8, 2006

Sponsored by
The USENIX Association

USENIX

In cooperation with ACM SIGOPS

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

The price is \$40 for members and \$50 for nonmembers. Outside the U.S.A. and Canada, please add \$15 per copy for postage (via air printed matter).

Thanks to Our Sponsors







Google



















The Network is the Computer™



Thanks to Our Media Sponsors

Addison-Wesley Professional/ Prentice Hall Professional ITtoolbox

Linux Journal

SNIA StorageNetworking.org Sys Admin

© 2006 by The USENIX Association All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

USENIX Association

Proceedings of the
7th USENIX Symposium on
Operating Systems
Design and Implementation
(OSDI '06)

November 6–8, 2006 Seattle, WA, USA

Symposium Organizers

Program Co-Chairs

Brian Bershad, University of Washington Jeff Mogul, Hewlett-Packard Labs

Program Committee

Martín Abadi, University of California, Santa Cruz, and Microsoft Research

Brad Calder, University of California, San Diego, and Microsoft

Brad Chen, Intel

Peter Druschel, Max Planck Institute for Software

Garth Gibson, Carnegie Mellon University and Panasas Derek McAuley, XenSource Inc.

Rob Pike, Google Inc.

Mema Roussopoulos, Harvard University Dawn Song, Carnegie Mellon University Chandu Thekkath, Microsoft Research Robbert van Renesse, Cornell University Jim Waldo, Sun Microsystems, Inc.

Bill Weihl, Google Inc.

Poster Session Chair

Mema Roussopoulos, Harvard University

Work-in-Progress Session Chair

Jim Waldo, Sun Microsystems, Inc.

Scribe for Program Committee Meeting

Paul Gauthier, University of Washington

Steering Committee

Eric Brewer, University of California, Berkeley Peter Chen, University of Michigan, Ann Arbor Mike Jones, Microsoft Jay Lepreau, University of Utah Ellie Young, USENIX

The USENIX Association Staff

External Reviewers

Greg Minshall

Michael Abd-El-Malek Andreas Haeberlen Atul Adya Marcos Aguilera Jonathan Aldrich Mary Baker Paul Barham Lujo Bauer Hans Boehm David Brumley Mihai Budiu Jeff Butler Miguel Castro Vincent Conitzer Russ Cox Marcel Dischinger Ed Lee Fred Douglis David Lie Úlfar Erlingsson Mark Luk Alexandra Fedorova Armando Fox Greg Ganger Debin Gao Andrew Goldberg Moises Goldszmidt

Ed Gronke

Mohammad Haghighat Benny Halevy Joe Halpern Maya Haridasan James Hendricks Rebecca Isaacs John Janakiraman Kevin Jeffay Chris Kantarjiev Terence Kelly Emre Kıcıman Chip Killian Hyang-Ah Kim John MacCormick Bruce Maggs Dahlia Malkhi Massimiliano Marcon Jonathan M. McCune Kirk McKusick Michael Mesnier

Alan Mislove Animesh Nandi Suman Nath Rolf Neugebauer James Newsome Bryan Parno Swapnil Patil Ramesh Peri David Petrou Ansley Post Venugopalan Ramasubramanian Mike Reiter Patrick Reynolds Thomas L. Rodeheffer Chris Sadler Asad Samar Stefan Savage Sebastian Schoenberg Bianca Schroeder Denis Serenyi Mehul Shah Anmol Sheth

Elaine Shi Atul Singh Craig Soules Suresh Srinivas Christopher Stewart Ion Stoica Doug Terry Eno Thereska Marc Unangst Amin Vahdat Geoff Voelker Einar Vollset Michael Vrable Brent Welch Matt Welsh Dan Wendlandt Udi Wieder Janet Wiener Ted Wobber Praveen Yalagandula Nickolai Zeldovich Lidong Zhou

7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)

November 6–8, 2006 Seattle, WA, USA

Index of Authorsv
Message from the Program Co-Chairs
Monday, November 6
Local Storage
Rethink the Sync
Type-Safe Disks
Stasis: Flexible Transactional Storage
Runtime Reliability Mechanisms
SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques
BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML
XFI: Software Guards for System Address Spaces
OS Implementation Strategies
Operating System Profiling via Latency Analysis
CRAMM: Virtual Memory Support for Garbage-Collected Applications
Flight Data Recorder: Monitoring Persistent-State Interactions to Improve Systems Management

Tuesday, November 7

Program Analysis Techniques
EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors
Securing Software by Enforcing Data-flow Integrity
From Uncertainty to Belief: Inferring the Specification Within
Distributed System Infrastructure
HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance
BAR Gossip
Bigtable: A Distributed Storage System for Structured Data
Distributed Systems of Little Things
EnsemBlue: Integrating Distributed Storage and Consumer Electronics
Persistent Personal Names for Globally Connected Mobile Devices
A Modular Network Layer for Sensornets
Operating System Structure
Making Information Flow Explicit in HiStar
Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable
Connection Handoff Policies for TCP Offload Network Interfaces

Wednesday, November 8

Distributed Storage and Locking
Ceph: A Scalable, High-Performance Distributed File System
Distributed Directory Service in the Farsite File System
The Chubby Lock Service for Loosely-Coupled Distributed Systems
Large Distributed Systems
Experiences Building PlanetLab
iPlane: An Information Plane for Distributed Services
Fidelity and Yield in a Volcano Monitoring Sensor Network

Index of Authors

Abadi, Martín	Gruber, Robert E	Nightingale, Edmund B 1
Alvisi, Lorenzo191	Harren, Matthew45	Peek, Daniel 219
Anderson, Thomas 367	Harris, Tim 147	Peterson, Larry
Anderson, Zachary 45	Howell, Jon321	Piatek, Michael 367
Back, Godmar161	Hsieh, Wilson C 205	Reis, Charles61
Bagrak, Ilya 45	Isdal, Tomas	Rhea, Sean 233
Bavier, Andy	Iyer, Rakesh 89	Rixner, Scott 293
Berger, Emery D 103	Johnson, Jeff	Rodrigues, Rodrigo177
Boyd-Wickizer, Silas 263	Joukov, Nikolai89	Roussev, Roussi 117
Brandt, Scott A 307	Kaashoek, Frans 233	Roy, Indrajit 191
Brewer, Eric 29, 45	Kaplan, Scott F 103	Sar, Can131
Budiu, Mihai	Kıcıman, Emre117	Sears, Russell 29
Burrows, Mike 205, 335	Kim, Hyong-youb293	Shenker, Scott249
Castro, Miguel 147	Kim, Sukun249	Shrira, Liuba177
Chandra, Tushar 205	Kohler, Eddie 263	Sivathanu, Gopalan15
Chang, Fay 205	Kremenek, Ted 161	Stoica, Ion249
Chen, Peter M 1	Krishnamurthy, Arvind 367	Strauss, Jacob 233
Clement, Allen 191	Kumar, Arunvijay 117	Sundararaman, Swaminathan 15
Condit, Jeremy 45	Lee, Juhan117	Ta-Min, Richard 279
Costa, Manuel 147	Lees, Jonathan	Tavakoli, Arsalan 249
Cowling, James	Lesniewski-Laas, Chris 233	Traeger, Avishay 89
Culler, David 249	Li, Harry C 191	Twohey, Paul
Dahlin, Michael191	Lie, David279	Veeraraghavan, Kaushik1
Daniels, Brad	Liskov, Barbara	Venkataramani, Arun367
Dean, Jeffrey205	Litty, Lionel 279	Verbowski, Chad117
Dixon, Colin	Long, Darrell D. E307	Vrable, Michael
Douceur, John	Lorincz, Konrad 381	Wallach, Deborah A 205
Dubrovsky, Opher	Lu, Shan	Wang, Helen J 61
Dunagan, John	Madhyastha, Harsha V 367	Wang, Yi-Min 117
Ee, Cheng Tien 249	Maltzahn, Carlos307	Wong, Edmund L 191
Engler, Dawson 131, 161	Mazières, David 263	Weil, Sage A 307
Ennals, Rob	Miller, Ethan L 307	Welsh, Matt
Erlingsson, Úlfar	Muir, Steve	Werner-Allen, Geoff381
Esmeir, Saher 61	Moon, Daekyeong249	Wright, Charles P
Fikes, Andrew	Morris, Robert 233	Yang, Junfeng131
Fiuczynski, Marc E351	Moss, J. Eliot B 103	Yang, Ting
Flinn, Jason	Myers, Daniel 177	Zadok, Erez 15, 89
Fonseca, Rodrigo 249	Napper, Jeff191	Zeldovich, Nickolai 263
Ford, Bryan	Necula, George C 45, 75	Zhou, Feng 45
Ghemawat Saniay 205	Ng. Andrew	

Message from the Program Co-Chairs

Dear friends.

We are very happy to welcome you to Seattle for the 7th USENIX OSDI conference. Nearly twelve years ago, OSDI was conceived as the "off-year SOSP." OSDI has always tried to attract and publish papers of the same depth and quality as those found in SOSP, but with somewhat more emphasis on implementation. We would like to think that OSDI is no longer a fill-in between SOSPs, but a true peer. You (and time) will tell whether OSDI has achieved that, but here we will say a little about how we ran the OSDI '06 process in support of this goal.

OSDI '06 received 155 submissions. The Call for Papers gave specific instructions for paper length and format. We asked six authors to withdraw their papers due to substantial non-compliance with these requirements, out of fairness to those authors who did follow the rules.

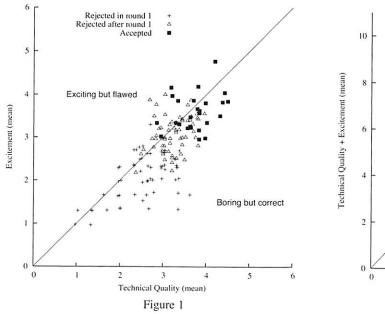
The remaining papers were reviewed in two rounds. In the first round, each paper received full reviews by three PC members, and many were also reviewed by an external reviewer. Based on the first-round reviews, the PC promoted 87 papers to the second round. Second-round papers received at least two additional full reviews by PC members, and many received additional PC or external reviews. This meant that every paper discussed at the PC meeting had been reviewed by at least five PC members. In all, we generated 697 reviews, most of them quite detailed.

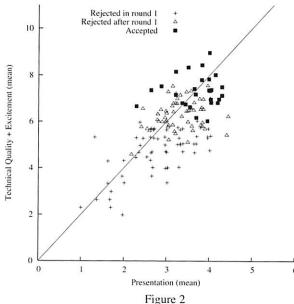
We accepted 27 of the 149 submissions, about 18%; this is in line with the historical average for OSDI and is consistent with the top conferences in the field. As is traditional for OSDI, we assigned a PC member to shepherd each accepted paper.

We allowed PC members to submit papers, but we judged these especially carefully. Of the five papers submitted by PC members, we accepted just one.

As program chairs, we were sensitive to the accusation that systems conferences often default to accepting papers that are clearly "correct," yet boring; this leads to conferences short on excitement, novelty, and controversy. Consequently, we nudged the OSDI reviewers to favor "exciting but flawed" submissions over "boring but correct" ones, as long as the flaws were minor and likely fixable. Reviewers assigned scores on a scale of 1–5 for "technical quality," "excitement," and "presentation quality."

After the program was set, we created scatter plots showing the relationship between these metrics and the final outcomes. Figure 1 shows that boring papers tended not to make it past the first round, but that exciting papers, even imperfect ones, tended to end up in the conference. Nevertheless, we ultimately rejected some high-excitement papers because we were not satisfied with their technical quality (see Figure 1).





Presentation quality also turned out (not surprisingly) to be an excellent predictor of acceptance. Most of the papers that were highly ranked in terms of presentation were ultimately accepted (see Figure 2). Good writing improves the reviewers' understanding of the technical content and makes it easier to appreciate the authors' own excitement.

As program chairs, we would like to thank all of the authors, reviewers, and conference attendees for their contributions, which help make this OSDI significant and memorable. We want to offer special thanks to the members of our program committee, each of whom voluntarily invested hundreds of hours to read, review, and discuss the submissions. We chose PC members not only for their technical skills but also for their dedication to making the right decisions; they did not let us down. It's been a lot of work, but we had fun.

Brian Bershad, University of Washington Jeff Mogul, Hewlett-Packard Labs Program Co-Chairs

Rethink the Sync

Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn Department of Electrical Engineering and Computer Science University of Michigan

Abstract

We introduce external synchrony, a new model for local file I/O that provides the reliability and simplicity of synchronous I/O, yet also closely approximates the performance of asynchronous I/O. An external observer cannot distinguish the output of a computer with an externally synchronous file system from the output of a computer with a synchronous file system. No application modification is required to use an externally synchronous file system: in fact, application developers can program to the simpler synchronous I/O abstraction and still receive excellent performance. We have implemented an externally synchronous file system for Linux, called xsyncfs. Xsyncfs provides the same durability and ordering guarantees as those provided by a synchronously mounted ext3 file system. Yet, even for I/O-intensive benchmarks, xsyncfs performance is within 7% of ext3 mounted asynchronously. Compared to ext3 mounted synchronously, xsyncfs is up to two orders of magnitude faster.

1 Introduction

File systems serve two opposing masters: durability and performance. The tension between these goals has led to two distinct models of file I/O: synchronous and asynchronous.

A synchronous file system (e.g., one mounted with the sync option on a Linux system) guarantees durability by blocking the calling application until modifications are committed to disk. Synchronous I/O provides a clean abstraction to users. Any file system operation they observe to complete is durable - data will not be lost due to a subsequent OS crash or power failure. Synchronous I/O also guarantees the ordering of modifications; if one operation causally precedes another, the effects of the second operation are never visible unless the effects of first operation are also visible. Unfortunately, synchronous I/O can be very slow because applications frequently block waiting for mechanical disk operations. In fact, our results show that blocking due to synchronous I/O can degrade the performance of disk-intensive benchmarks by two orders of magnitude.

In contrast, an asynchronous file system does not block the calling application, so modifications are typically committed to disk long after the call completes. This is fast, but not safe. Users view output that depends on uncommitted modifications. If the system crashes or loses power before those modifications commit, the output observed by the user was invalid. Asynchronous I/O also complicates applications that require durability or ordering guarantees. Programmers must insert explicit synchronization operations such as fsync to enforce the guarantees required by their applications. They must sometimes implement complex group commit strategies to achieve reasonable performance. Despite the poor guarantees provided to users and programmers, most local file systems provide an asynchronous I/O abstraction by default because synchronous I/O is simply too slow.

The tension between durability and performance leads to surprising behavior. For instance, on most desktop operating systems, even executing an explicit synchronization command such as fsync does not protect against data loss in the event of a power failure [13]. This behavior is not a bug, but rather a conscious design decision to sacrifice durability for performance [27]. For example, on fsync, the Linux 2.4 kernel commits data to the volatile hard drive cache rather than to the disk platter. If a power failure occurs, the data in the drive cache is lost. Because of this behavior, applications that require stronger durability guarantees, such as the MySQL database, recommend disabling the drive cache [15]. While MacOS X and the Linux 2.6 kernel provide mechanisms to explicitly flush the drive cache, these mechanisms are not enabled by default due to the severe performance degradation they can cause.

We show that a new model of local file I/O, which we term *external synchrony*, resolves the tension between durability and performance. External synchrony provides the reliability and simplicity of synchronous I/O, while closely approaching the performance of asynchronous I/O. In external synchrony, we view the abstraction of synchronous I/O as a set of guarantees that are provided to the clients of the file system. In con-

trast to asynchronous I/O, which improves performance by substantially weakening these guarantees, externally synchronous I/O provides the same guarantees, but it changes the clients to which the guarantees are provided.

Synchronous I/O reflects the *application-centric* view of modern operating systems. The return of a synchronous file system call guarantees durability to the application since the calling process is blocked until modifications commit. In contrast, externally synchronous I/O takes a *user-centric* view in which it guarantees durability not to the application, but to any external entity that observes application output. An externally synchronous system returns control to the application before committing data. However, it subsequently buffers all output that causally depends on the uncommitted modification. Buffered output is only externalized (sent to the screen, network, or other external device) after the modification commits.

From the viewpoint of an external observer such as a user or an application running on another computer, the guarantees provided by externally synchronous I/O are identical to the guarantees provided by a traditional file system mounted synchronously. An external observer never sees output that depends on uncommitted modifications. Since external synchrony commits modifications to disk in the order they are generated by applications, an external observer will not see a modification unless all other modifications that causally precede that modification are also visible. However, because externally synchronous I/O rarely blocks applications, its performance approaches that of asynchronous I/O.

Our externally synchronous Linux file system, xsyncfs, uses mechanisms developed as part of the Speculator project [17]. When a process performs a synchronous I/O operation, xsyncfs validates the operation, adds the modifications to a file system transaction, and returns control to the calling process without waiting for the transaction to commit. However, xsyncfs also taints the calling process with a commit dependency that specifies that the process is not allowed to externalize any output until the transaction commits. If the process writes to the network, screen, or other external device, its output is buffered by the operating system. The buffered output is released only after all disk transactions on which the output depends commit. If a process with commit dependencies interacts with another process on the same computer through IPC such as pipes, the file cache, or shared memory, the other process inherits those dependencies so that it also cannot externalize output until the transaction commits. The performance of xsyncfs is generally quite good since applications can perform computation and initiate further I/O operations while waiting for a transaction to commit. In most cases, output is delayed by no more than the time to commit a single transaction — this is typically less than the perception threshold of a human user.

Xsyncfs uses *output-triggered commits* to balance throughput and latency. Output-triggered commits track the causal relationship between external output and file system modifications to decide when to commit data. Until some external output is produced that depends upon modified data, xsyncfs may delay committing data to optimize for throughput. However, once some output is buffered that depends upon an uncommitted modification, an immediate commit of that modification is triggered to minimize latency for any external observer.

Our results to date are very positive. For I/O intensive benchmarks such as Postmark and an Andrew-style build, the performance of xsyncfs is within 7% of the default asynchronous implementation of ext3. Compared to current implementations of synchronous I/O in the Linux kernel, external synchrony offers better performance and better reliability. Xsyncfs is up to an order of magnitude faster than the default version of ext3 mounted synchronously, which allows data to be lost on power failure because committed data may reside in the volatile hard drive cache. Xsyncfs is up to two orders of magnitude faster than a version of ext3 that guards against losing data on power failure. Xsyncfs sometimes even improves the performance of applications that do their own custom synchronization. Running on top of xsyncfs, the MySQL database executes a modified version of the TPC-C benchmark up to three times faster than when it runs on top of ext3 mounted asynchronously.

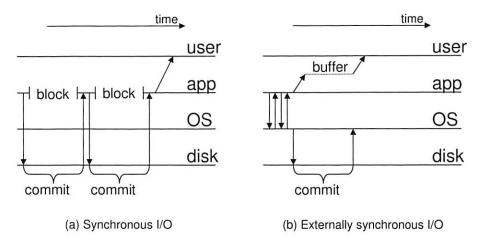
2 Design overview

2.1 Principles

The design of external synchrony is based on two principles. First, we define externally synchronous I/O by its externally observable behavior rather than by its implementation. Second, we note that application state is an internal property of the computer system. Since application state is not directly observable by external entities, the operating system need not treat changes to application state as an external output.

Synchronous I/O is usually defined by its implementation: an I/O is considered synchronous if the calling application is blocked until after the I/O completes [26]. In contrast, we define externally synchronous I/O by its observable behavior: we say that an I/O is externally synchronous if the external output produced by the computer system cannot be distinguished from output that could have been produced if the I/O had been synchronous.

The next step is to precisely define what is considered external output. Traditionally, the operating system takes



This figure shows the behavior of a sample application that makes two file system modifications, then displays output to an external device. The diagram on the left shows how the application executes when its file I/O is synchronous; the diagram on the right shows how it executes when its file I/O is externally synchronous.

Figure 1. Example of externally synchronous file I/O

an *application-centric* view of the computer system, in which it considers applications to be external entities observing its behavior. This view divides the computer system into two partitions: the kernel, which is considered internal state, and the user level, which is considered external state. Using this view, the return from a system call is considered an externally visible event.

However, users, not applications, are the true observers of the computer system. Application state is only visible through output sent to external devices such as the screen and network. By regarding application state as internal to the computer system, the operating system can take a *user-centric* view in which only output sent to an external device is considered externally visible. This view divides the computer system into three partitions, the kernel and applications, both of which are considered internal state, and the external interfaces, which are considered externally visible. Using this view, changes to application state, such as the return from a system call, are not considered externally visible events.

The operating system can implement user-centric guarantees because it controls access to external devices. Applications can only generate external events with the cooperation of the operating system. Applications must invoke this cooperation either directly by making a system call or indirectly by mapping an externally visible device.

2.2 Correctness

Figure 1 illustrates these principles by showing an example single-threaded application that makes two file system modifications and writes some output to the screen. In the diagram on the left, the file modifications made by the application are synchronous. Thus, the application blocks until each modification commits.

We say that external output of an externally synchronous system is equivalent to the output of a synchronous one if (a) the values of the external outputs are the same, and (b) the outputs occur in the same causal order, as defined by Lamport's *happens before* relation [9]. We consider disk commits external output because they change the stable image of the file system. If the system crashes and reboots, the change to the stable image is visible. Since the operating system cannot control when crashes occur, it must treat disk commits as external output. Thus, in Figure 1(a), there are three external outputs: the two commits and the message displayed on the screen.

An externally synchronous file I/O returns the same result to applications that would have been returned by a synchronous I/O. The file system does all processing that would be done for a synchronous I/O, including validation and changing the volatile (in-memory) state of the file system, except that it does not actually commit the modification to disk before returning. Because the results that an application sees from an externally synchronous I/O are equivalent to the results it would have seen if the I/O had been synchronous, the external output it produces is the same in both cases.

An operating system that supports external synchrony must ensure that external output occurs in the same causal order that would have occurred had I/O been performed synchronously. Specifically, if an external output causally follows an externally synchronous file I/O, then that output cannot be observed before the file I/O has been committed to disk. In the example, this means that the second file modification made by the application cannot commit before the first, and that the screen output cannot be seen before both modifications commit.

2.3 Improving performance

The externally synchronous system in Figure 1(b) makes two optimizations to improve performance. First, the two modifications are group committed as a single file system transaction. Because the commit is atomic, the effects of the second modification are never seen unless the effects of the first are also visible. Grouping multiple modifications into one transaction has many benefits: the commit of all modifications is done with a single sequential disk write, writes to the same disk block are coalesced in the log, and no blocks are written to disk at all if data writes are closely followed by deletion. For example, ext3 employs value logging — when a transaction commits, only the latest version of each block is written to the journal. If a temporary file is created and deleted within a single transaction, none of its blocks are written to disk. In contrast, a synchronous file system cannot group multiple modifications for a single-threaded application because the application does not begin the second modification until after the first commits.

The second optimization is buffering screen output. The operating system must delay the externalization of the output until after the commit of the file modifications to obey the causal ordering constraint of externally synchronous I/O. One way to enforce this ordering would be to block the application when it initiates external output. However, the asynchronous nature of the output enables a better solution. The operating system instead buffers the output and allows the process that generated the output to continue execution. After the modifications are committed to disk, the operating system releases the output to the device for which it was destined.

This design requires that the operating system track the causal relationship between file system modifications and external output. When a process writes to the file system, it inherits a commit dependency on the uncommitted data that it wrote. When a process with commit dependencies modifies another kernel object (process, pipe, file, UNIX socket, etc.) by executing a system call, the operating system marks the modified objects with the same commit dependencies. Similarly, if a process observes the state of another kernel object with commit dependencies, the process inherits those dependencies. If a process with commit dependencies executes a system call for which the operating system cannot track the flow of causality (e.g., an ioctl), the process is blocked until its file systems modifications have been committed. Any external output inherits the commit dependencies of the process that generated it — the operating system buffers the output until the last dependency is resolved by committing modifications to disk.

2.4 Deciding when to commit

An externally synchronous file system uses the causal relationship between external output and file modifications to trigger commits. There is a well-known tradeoff between throughput and latency for group commit strategies. Delaying a group commit in the hope that more modifications will occur in the near future can improve throughput by amortizing more modifications across a single commit. However, delaying a commit also increases latency — in our system, commit latency is especially important because output cannot be externalized until the commit occurs.

Latency is unimportant if no external entity is observing the result. Specifically, until some output is generated that causally depends on a file system transaction, committing the transaction does not change the observable behavior of the system. Thus, the operating system can improve throughput by delaying a commit until some output that depends on the transaction is buffered (or until some application that depends on the transaction blocks due to an ioctl or similar system call). We call this strategy *output-triggered commits* since the attempt to generate output that is causally dependent upon modifications to be written to disk triggers the commit of those modifications.

Output-triggered commits enable an externally synchronous file system to maximize throughput when output is not being displayed (for example, when it is piped to a file). However, when a user could be actively observing the results of a transaction, commit latency is small.

2.5 Limitations

One potential limitation of external synchrony is that it complicates application-specific recovery from catastrophic media failure because the application continues execution before such errors are detected. Although the kernel validates each modification before writing it to the file cache, the physical write of the data to disk may subsequently fail. While smaller errors such as a bad disk block are currently handled by the disk or device driver, a catastrophic media failure is rarely masked at these levels. Theoretically, a file system mounted synchronously could propagate such failures to the application. However, a recent survey of common file systems [20] found that write errors are either not detected by the file system (ext3, jbd, and NTFS) or induce a kernel panic (ReiserFS). An externally synchronous file system could propagate failures to applications by using Speculator to checkpoint a process before it modifies the file system. If a catastrophic failure occurs, the process would be rolled back and notified of the failure. We rejected this solution because it would both greatly increase the complexity of external synchrony and severely penalize its performance. Further, it is unclear that catastrophic failures are best handled by applications — it seems best to handle them in the operating system, either by inducing a kernel panic or (preferably) by writing data elsewhere.

Another limitation of external synchrony is that the user may have some temporal expectations about when modifications are committed to disk. As defined so far, an externally synchronous file system could indefinitely delay committing data written by an application with no external output. If the system crashes, a substantial amount of work could be lost. Xsyncfs therefore commits data every 5 seconds, even if no output is produced. The 5 second commit interval is the same value used by ext3 mounted asynchronously.

A final limitation of external synchrony is that modifications to data in two different file systems cannot be easily committed with a single disk transaction. Potentially, we could share a common journal among all local file systems, or we could implement a two-phase commit strategy. However, a simpler solution is to block a process with commit dependencies for one file system before it modifies data in a second. Speculator would map each dependency to a specific file system. When a process writes to a file system, Speculator would verify that the process depends only on the file system it is modifying; if it depends on another file system, Speculator would block it until its previous modifications commit.

3 Implementation

3.1 External synchrony

We next provide a brief overview of Speculator [17] and how it supports externally synchronous file systems.

3.1.1 Speculator background

Speculator improves the performance of distributed file systems by hiding the performance cost of remote operations. Rather than block during a remote operation, a file system predicts the operation's result, then uses Speculator to checkpoint the state of the calling process and speculatively continue its execution based on the predicted result. If the prediction is correct, the checkpoint is discarded; if it is incorrect, the calling process is restored to the checkpoint, and the operation is retried.

Speculator adds two new data structures to the kernel. A *speculation* object tracks all process and kernel state that depends on the success or failure of a speculative operation. Each speculative object in the kernel has an *undo log* that contains the state needed to undo speculative modifications to that object. As processes interact with kernel objects by executing system calls, Speculator

uses these data structures to track causal dependencies. For example, if a speculative process writes to a pipe, Speculator creates an entry in the pipe's undo log that refers to the speculations on which the writing process depends. If another process reads from the pipe, Speculator creates an undo log entry for the reading process that refers to all speculations on which the pipe depends.

Speculator ensures that speculative state is never visible to an external observer. If a speculative process executes a system call that would normally externalize output, Speculator buffers its output until the outcome of the speculation is decided. If a speculative process performs a system call that Speculator is unable to handle by either transferring causal dependencies or buffering output, Speculator blocks it until it becomes non-speculative.

3.1.2 From speculation to synchronization

Speculator ties dependency tracking and output buffering to other features, such as checkpoint and rollback, that are not needed to support external synchrony. Worse yet, these unneeded features come at a substantial performance cost. This led us to factor out the functionality in Speculator common to both speculative execution and external synchrony. We modified the Speculator interface to allow each file system to specify the additional Speculator features that it requires. This allows a single computer to run both a speculative distributed file system and an externally synchronous local file system.

Both speculative execution and external synchrony enforce restrictions on when external output may be observed. Speculative execution allows output to be observed based on *correctness*; output is externalized after all speculations on which that output depends have proven to be correct. In contrast, external synchrony allows output to be observed based on *durability*; output is externalized after all file system operations on which that output depends have been committed to disk.

In external synchrony, a commit dependency represents the causal relationship between kernel state and an uncommitted file system modification. Any kernel object that has one or more associated commit dependencies is referred to as *uncommitted*. Any external output from a process that is uncommitted is buffered within the kernel until the modifications on which the output depends have been committed. In other words, uncommitted output is never visible to an external observer.

When a process writes to an externally synchronous file system, Speculator marks the process as uncommitted. It also creates a commit dependency between the process and the uncommitted file system transaction that contains the modification. When the file system commits the transaction to disk, the commit dependency is removed.

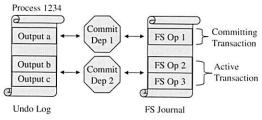
Once all commit dependencies for buffered output have been removed, Speculator releases that output to the external device to which it was written. When the last commit dependency for a process is discarded, Speculator marks the process as committed.

Speculator propagates commit dependencies among kernel objects and processes using the same mechanisms it uses to propagate speculative dependencies. However, since external synchrony does not require checkpoint and rollback, the propagation of dependencies is considerably easier to implement. For instance, before a process inherits a new speculative dependency, Speculator must checkpoint its state with a copy-on-write fork. In contrast, when a process inherits a commit dependency, no checkpoint is needed since the process will never be rolled back. To support external synchrony, Speculator maintains the same many-to-many relationship between commit dependencies and undo logs as it does for speculations and undo logs. Since commit dependencies are never rolled back, undo logs need not contain data to undo the effects of an operation. Therefore, undo logs in an externally synchronous system only track the relationship between commit dependencies and kernel objects and reveal which buffered output can be safely released. This simplicity enables Speculator to support more forms of interaction among uncommitted processes than it supports for speculative processes. For example, checkpointing multi-threaded processes for speculative execution is a thorny problem [17, 21]. However, as discussed in Section 3.5, tracking their commit dependencies is substantially simpler.

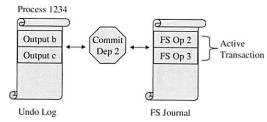
3.2 File system support for external synchrony

We modified ext3, a journaling Linux file system, to create xsyncfs. In its default ordered mode, ext3 writes only metadata modifications to its journal. In its journaled mode, ext3 writes both data and metadata modifications. Modifications from many different file system operations may be grouped into a single compound journal transaction that is committed atomically. Ext3 writes modifications to the active transaction — at most one transaction may be active at any given time. A commit of the active transaction is triggered when journal space is exhausted, an application performs an explicit synchronization operation such as fsync, or the oldest modification in the transaction is more than 5 seconds old. After the transaction starts to commit, the next modification triggers the creation of a new active transaction. Only one transaction may be committing at any given time, so the next transaction must wait for the commit of the prior transaction to finish before it commits.

Figure 2 shows how the external synchrony data structures change when a process interacts with xsyncfs. In



(a) Data structures with a committing and active transaction



(b) Data structures after the first transaction commits

Figure 2. The external synchrony data structures

Figure 2(a), process 1234 has completed three file system operations, sending output to the screen after each one. Since the output after the first operation triggered a transaction commit, the two following operations were placed in a new active transaction. The output is buffered in the undo log; the commit dependencies maintain the relationship between buffered output and uncommitted data. In Figure 2(b), the first transaction has been committed to disk. Therefore, the output that depended upon the committed transaction has been released to the screen and the commit dependency has been discarded.

Xsyncfs uses journaled mode rather than the default ordered mode. This change guarantees ordering; specifically, the property that if an operation A causally precedes another operation B, the effects of B should never be visible unless the effects of A are also visible. This guarantee requires that B never be committed to disk before A. Otherwise, a system crash or power failure may occur between the two commits - in this case, after the system is restarted, B will be visible when A is not. Since journaled mode adds all modifications for A to the journal before the operation completes, those modifications must already be in the journal when B begins (since B causally follows A). Thus, either B is part of the same transaction as A (in which case the ordering property holds since A and B are committed atomically), or the transaction containing A is already committed before the transaction containing B starts to commit.

In contrast, the default mode in ext3 does not provide ordering since data modifications are not journaled. The kernel may write the dirty blocks of A and B to disk in any order as long as the data reaches disk before the metadata in the associated journal transaction commits. Thus, the data modifications for B may be visible after a crash without the modifications for A being visible.

Xsyncfs informs Speculator when a new journal transaction is created — this allows Speculator to track state that depends on the uncommitted transaction. Xsyncfs also informs Speculator when a new modification is added to the transaction and when the transaction commits.

As described in Section 1, the default behavior of ext3 does not guarantee that modifications are durable after a power failure. In the Linux 2.4 kernel, durability can be ensured only by disabling the drive cache. The Linux 2.6.11 kernel provides the option of using *write barriers* to flush the drive cache before and after writing each transaction commit record. Since Speculator runs on a 2.4 kernel, we ported write barriers to our kernel and modified xsyncfs to use write barriers to guarantee that all committed modifications are preserved, even on power failure.

3.3 Output-triggered commits

Xsyncfs uses the causal relationship between disk I/O and external output to balance the competing concerns of throughput and latency. Currently, ext3 commits its journal every 5 seconds, which typically groups the commit of many file system operations. This strategy optimizes for throughput, a logical behavior when writes are asynchronous. However, latency is an important consideration in xsyncfs since users must wait to view output until the transactions on which that output depends commit. If xsyncfs were to use the default ext3 commit strategy, disk throughput would be high, but the user might be forced to wait up to 5 seconds to see output. This behavior is clearly unacceptable for interactive applications.

We therefore modified Speculator to support outputtriggered commits. Speculator provides callbacks to xsyncfs when it buffers output or blocks a process that performed a system call for which it cannot track the propagation of causal dependencies (e.g., an ioct1). Xsyncfs uses the ext3 strategy of committing every 5 seconds unless it receives a callback that indicates that Speculator blocked or buffered output from a process that depends on the active transaction. The receipt of a callback triggers a commit of the active transaction.

Output-triggered commits adapt the behavior of the file system according to the observable behavior of the system. For instance, if a user directs output from a running application to the screen, latency is reduced by committing transactions frequently. If the user instead redirects the output to a file, xsyncfs optimizes for throughput by committing every 5 seconds. Optimizing for throughput is correct in this instance since the only event the user can observe is the completion of the application (and

the completion would trigger a commit if it is a visible event). Finally, if the user were to observe the contents of the file using a different application, e.g., tail, xsyncfs would correctly optimize for latency because Speculator would track the causal relationship through the kernel data structures from tail to the transaction and provide callbacks to xsyncfs. When tail attempts to output data to the screen, Speculator callbacks will cause xsyncfs to commit the active transaction.

3.4 Rethinking sync

Asynchronous file systems provide explicit synchronization operations such as sync and fdatasync for applications with durability or ordering constraints. In a synchronous file system, such synchronization operations are redundant since ordering and durability are already guaranteed for all file system operations. However, in an externally synchronous file system, some extra support is needed to minimize latency. For instance, a user who types "sync" in a terminal would prefer that the command complete as soon as possible.

When xsyncfs receives a synchronization call such as sync from the VFS layer, it creates a commit dependency between the calling process and the active transaction. Since this does not require a disk write, the return from the synchronization call is almost instantaneous. If a visible event occurs, such as the completion of the sync process, Speculator will issue a callback that causes xsyncfs to commit the active transaction.

External synchrony simplifies the file system abstraction. Since xsyncfs requires no application modification, programmers can write the same code that they would write if they were using a unmodified file system mounted synchronously. They do not need explicit synchronization calls to provide ordering and durability since xsyncfs provides these guarantees by default for all file system operations. Further, since xsyncfs does not incur the large performance penalty usually associated with synchronous I/O, programmers do not need complicated group commit strategies to achieve acceptable performance. Group commit is provided transparently by xsyncfs.

Of course, a hand-tuned strategy might offer better performance than the default policies provided by xsyncfs. However, as described in Section 3.3, there are some instances in which xsyncfs can optimize performance when an application solution cannot. Since xsyncfs uses output-triggered commits, it knows when no external output has been generated that depends on the current transaction; in these instances, xsyncfs uses group commit to optimize throughput. In contrast, an application-specific commit strategy cannot determine the visibility

of its actions beyond the scope of the currently executing process; it must therefore conservatively commit modifications before producing external messages.

For example, consider a client that issues two sequential transactions to a database server on the same computer and then produces output. Xsyncfs can safely group the commit of both transactions. However, the database server (which does not use output-triggered commits) must commit each transaction separately since it cannot know whether or not the client will produce output after it is informed of the commit of the first transaction.

3.5 Shared memory

Speculator does not propagate speculative dependencies when processes interact through shared memory due to the complexity of checkpointing at arbitrary states in a process' execution. Since commit dependencies do not require checkpoints, we enhanced Speculator to propagate them among processes that share memory.

Speculator can track causal dependencies because processes can only interact through the operating system. Usually, this interaction involves an explicit system call (e.g., write) that Speculator can intercept. However, when processes interact through shared memory regions, only the sharing and unsharing of regions is visible to the operating system. Thus, Speculator cannot readily intercept individual reads and writes to shared memory.

We considered marking a shared memory page inaccessible when a process with write permission inherits a commit dependency that a process with read permission does not have. This would trigger a page fault whenever a process reads or writes the shared page. If a process reads the page after another writes it, any commit dependencies would be transferred from the writer to the reader. Once these processes have the same commit dependencies, the page can be restored to its normal protections. We felt this mechanism would perform poorly because of the time needed to protect and unprotect pages, as well as the extra page faults that would be incurred.

Instead, we decided to use an approach that imposes less overhead but might transfer dependencies when not strictly necessary. We make a conservative assumption that processes with write permission for a shared memory region are continually writing to that region, while processes with read permission are continually reading it. When a process with write permission for a shared region inherits a new commit dependency, any process with read permission for that region atomically inherits the same dependency.

Speculator uses the same mechanism to track commit dependencies transfered through memory-mapped files. Similarly, Speculator is conservative when propagating dependencies for multi-threaded applications — any dependency inherited by one thread is inherited by all.

4 Evaluation

Our evaluation answers the following questions:

- How does the durability of xsyncfs compare to current file systems?
- How does the performance of xsyncfs compare to current file systems?
- How does xsyncfs affect the performance of applications that synchronize explicitly?
- How much do output-triggered commits improve the performance of xsyncfs?

4.1 Methodology

All computers used in our evaluation have a 3.02 GHZ Pentium 4 processor with 1 GB of RAM. Each computer has a single Western Digital WD-XL40 hard drive, which is a 7200 RPM 120 GB ATA 100 drive with a 2 MB ondisk cache. The computers run Red Hat Enterprise Linux version 3 (kernel version 2.4.21). We use a 400 MB journal size for both ext3 and xsyncfs. For each benchmark, we measured ext3 executing in both journaled and ordered mode. Since journaled mode executed faster in every benchmark, we report only journaled mode results in this evaluation. Finally, we measured the performance of ext3 both using write barriers and with the drive cache disabled. In all cases write barriers were faster than disabling the drive cache since the drive cache improves read times and reduces the frequency of writes to the disk platter. Thus, we report only results using write barriers.

4.2 Durability

Our first benchmark empirically confirms that without write barriers, ext3 does not guarantee durability. This result holds in both journaled and ordered mode, whether ext3 is mounted synchronously or asynchronously, and even if fsync commands are issued by the application after every write. Even worse, our results show that, despite the use of journaling in ext3, a loss of power can corrupt data and metadata stored in the file system.

We confirmed these results by running an experiment in which a test computer continuously writes data to its local file system. After each write completes, the test computer sends a UDP message that is logged by a remote computer. During the experiment, we cut power to the test computer. After it reboots, we compare the state of its file system to the log on the remote computer.

File system configuration	Data durable on write	Data durable on fsync
Asynchronous	No	Not on power failure
Synchronous	Not on power failure	Not on power failure
Synchronous with write barriers	Yes	Yes
External synchrony	Yes	Yes

This figure describes whether each file system provides durability to the user when an application executes a write or fsync system call. A "Yes" indicates that the file system provides durability if an OS crash or power failure occurs.

Figure 3. When is data safe?

Our goal was to determine when each file system guarantees durability and ordering. We say a file system fails to provide durability if the remote computer logs a message for a write operation, but the test computer is missing the data written by that operation. In this case, durability is not provided because an external observer (the remote computer) saw output that depended on data that was subsequently lost. We say a file system fails to provide ordering if the state of the file after reboot violates the temporal ordering of writes. Specifically, for each block in the file, ordering is violated if the file does not also contain all previously-written blocks.

For each configuration shown in Figure 3, we ran four trials of this experiment: two in journaled mode and two in ordered mode. As expected, our results confirm that ext3 provides durability only when write barriers are used. Without write barriers, synchronous operations ensure only that modifications are written to the hard drive cache. If power fails before the modifications are written to the disk platter, those modifications are lost.

Some of our experiments exposed a dangerous behavior in ext3: unless write barriers are used, power failures can corrupt both data and metadata stored on disk. In one experiment, a block in the file being modified was silently overwritten with garbage data. In another, a substantial amount of metadata in the file system, including the superblock, was overwritten with garbage. In the latter case, the test machine failed to reboot until the file system was manually repaired. In both cases, corruption is caused by the commit block for a transaction being written to the disk platter before all data blocks in that transaction are written to disk. Although the operating system wrote the blocks to the drive cache in the correct order, the hard drive reorders the blocks when writing them to the disk platter. After this happens, the transaction is committed during recovery even though several data blocks do not contain valid data. Effectively, this overwrites disk blocks with uninitialized data.

Our results also confirm that ext3 without write barriers writes data to disk out of order. Journaled mode alone is insufficient to provide ordering since the order of writing transactions to the disk platter may differ from the order of writing transactions to the drive cache. In contrast,

ext3 provides both durability and ordering when write barriers are combined with some form of synchronous operation (either mounting the file system synchronously or calling fsync after each modification). If write barriers are not available, the equivalent behavior could also be achieved by disabling the hard drive cache.

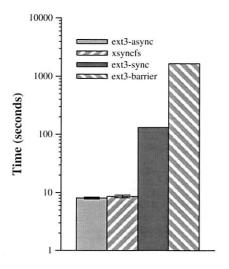
The last row of Figure 3 shows results for xsyncfs. As expected, xsyncfs provides both durability and ordering.

4.3 The PostMark benchmark

We next ran the PostMark benchmark, which was designed to replicate the small file workloads seen in electronic mail, netnews, and web-based commerce [8]. We used PostMark version 1.5, running in a configuration that creates 10,000 files, performs 10,000 transactions consisting of file reads, writes, creates, and deletes, and then removes all files. The PostMark benchmark has a single thread of control that executes file system operations as quickly as possible. PostMark is a good test of file system throughput since it does not generate any output or perform any substantial computation.

Each bar in Figure 4 shows the time to complete the Post-Mark benchmark. The y-axis is logarithmic because of the substantial slowdown of synchronous I/O. The first bar shows results when ext3 is mounted asynchronously. As expected, this offers the best performance since the file system buffers data in memory up to 5 seconds before writing it to disk. The second bar shows results using xsyncfs. Despite the I/O intensive nature of PostMark, the performance of xsyncfs is within 7% of the performance of ext3 mounted asynchronously. After examining the performance of xsyncfs, we determined that the overhead of tracking causal dependencies in the kernel accounts for most of the difference.

The third bar shows performance when ext3 is mounted synchronously. In this configuration the writing process is blocked until its modifications are committed to the drive cache. Ext3 in synchronous mode is over an order of magnitude slower than xsyncfs, even though xsyncfs provides stronger durability guarantees. Throughput is limited by the size of the drive cache; once the cache fills, subsequent writes block until some data in the cache is written to the disk platter.



This figure shows the time to run the PostMark benchmark — the y-axis is logarithmic. Each value is the mean of 5 trials — the (relatively small) error bars are 90% confidence intervals.

Figure 4. The PostMark file system benchmark

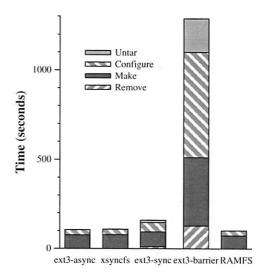
The last bar in Figure 4 shows the time to complete the benchmark when ext3 is mounted synchronously and write barriers are used to prevent data loss when a power failure occurs. Since write barriers synchronously flush the drive cache twice for each file system transaction, ext3's performance is over two orders of magnitude slower than that of xsyncfs.

Due to the high cost of durability, high end storage systems sometimes use specialized hardware such as a non-volatile cache to improve performance [7]. This eliminates the need for write barriers. However, even with specialized hardware, we expect that the performance of ext3 mounted synchronously would be no better than the third bar in Figure 4, which writes data to a volatile cache. Thus, use of xsyncfs should still lead to substantial performance improvements for synchronous operations even when the hard drive has a non-volatile cache of the same size as the volatile cache on our drive.

4.4 The Apache build benchmark

We next ran a benchmark in which we untar the Apache 2.0.48 source tree into a file system, run configure in an object directory within that file system, run make in the object directory, and remove all files. The Apache build benchmark requires the file system to balance throughput and latency; it displays large amounts of screen output interleaved with disk I/O and computation.

Figure 5 shows the total amount of time to run the benchmark, with shadings within each bar showing the time for each stage. Comparing the first two bars in the graph, xsyncfs performs within 3% of ext3 mounted asynchronously. Since xsyncfs releases output as soon



This figure shows the time to run the Apache build benchmark. Each value is the mean of 5 trials — the (relatively small) error bars are 90% confidence intervals.

Figure 5. The Apache build benchmark

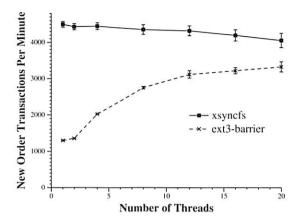
as the data on which it depends commits, output appears promptly during the execution of the benchmark.

For comparison, the bar at the far right of the graph shows the time to execute the benchmark using a memory-only file system, RAMFS. This provides a lower bound on the performance of a local file system, and it isolates the computation requirements of the benchmark. Removing disk I/O by running the benchmark in RAMFS improves performance by only 8% over xsyncfs because the remainder of the benchmark is dominated by computation.

The third bar in Figure 5 shows that ext3 mounted in synchronous mode is 46% slower than xsyncfs. Since computation dominates I/O in this benchmark, any difference in I/O performance is a smaller part of overall performance. The fourth bar shows that ext3 mounted synchronously with write barriers is over 11 times slower than xsyncfs. If we isolate the cost of I/O by subtracting the cost of computation (calculated using the RAMFS result), ext3 mounted synchronously is 7.5 times slower than xsyncfs while ext3 mounted synchronously with write barriers is over two orders of magnitude slower than xsyncfs. These isolated results are similar to the values that we saw for the PostMark experiments.

4.5 The MySQL benchmark

We were curious to see how xsyncfs would perform with an application that implements its own group commit strategy. We therefore ran a modified version of the OSDL TPC-C benchmark [18] using MySQL version 5.0.16 and the InnoDB storage engine. Since both MySQL and the TPC-C benchmark client are



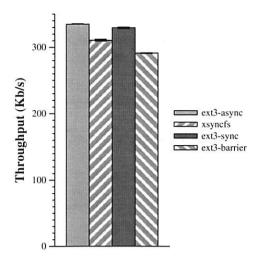
This figure shows the New Order Transactions Per Minute when running a modified TPC-C benchmark on MySQL with varying numbers of clients. Each result is the mean of 5 trials — the error bars are 90% confidence intervals.

Figure 6. The MySQL benchmark

multi-threaded, this benchmark measures the efficacy of xsyncfs's support for shared memory. TPC-C measures the New Order Transactions Per Minute (NOTPM) a database can process for a given number of simultaneous client connections. The total number of transactions performed by TPC-C is approximately twice the number of New Order Transactions. TPC-C requires that a database provide ACID semantics, and MySOL requires either disabling the drive cache or using write barriers to provide durability. Therefore, we compare xsyncfs with ext3 mounted asynchronously using write barriers. Since the client ran on the same machine as the server, we modified the benchmark to use UNIX sockets. This allows xsyncfs to propagate commit dependencies between the client and server on the same machine. In addition, we modified the benchmark to saturate the MySQL server by removing any wait times between transactions and creating a data set that fits completely in memory.

Figure 6 shows the NOTPM achieved as the number of clients is increased from 1 to 20. With a single client, MySQL completes 3 times as many NOTPM using xsyncfs. By propagating commit dependencies to both the MySQL server and the requesting client, xsyncfs can group commit transactions from a single client, significantly improving performance. In contrast, MySQL cannot benefit from group commit with a single client because it must conservatively commit each transaction before replying to the client.

When there are multiple clients, MySQL can group the commit of transactions from different clients. As the number of clients grows, the gap between xsyncfs and ext3 mounted asynchronously with write barriers shrinks. With 20 clients, xsyncfs improves TPC-C performance by 22%. When the number of clients reaches 32, the performance of ext3 mounted asynchronously



This figure shows the mean throughput achieved when running the SPECweb99 benchmark with 50 simultaneous connections. Each result is the mean of three trials, with error bars showing the highest and lowest result.

Figure 7. Throughput in the SPECweb99 benchmark

with write barriers matches the performance of xsyncfs. From these results, we conclude that even applications such as MySQL that use a custom group commit strategy can benefit from external synchrony if the number of concurrent transactions is low to moderate.

Although ext3 mounted asynchronously without write barriers does not meet the durability requirements for TPC-C, we were still curious to see how its performance compared to xsyncfs. With only 1 or 2 clients, MySQL executes 11% more NOTPM with xsyncfs than it executes with ext3 without write barriers. With 4 or more clients, the two configurations yield equivalent performance within experimental error.

4.6 The SPECweb99 benchmark

Since our previous benchmarks measured only work-loads confined to a single computer, we also ran the SPECweb99 [29] benchmark to examine the impact of external synchrony on a network-intensive application. In the SPECweb99 benchmark, multiple clients issue a mix of HTTP GET and POST requests. HTTP GET requests are issued for both static and dynamic content up to 1 MB in size. A single client, emulating 50 simultaneous connections, is connected to the server, which runs Apache 2.0.48, by a 100 Mb/s Ethernet switch. As we use the default Apache settings, 50 connections are sufficient to saturate our server.

We felt that this benchmark might be especially challenging for xsyncfs since sending a network message externalizes state. Since xsyncfs only tracks causal dependencies on a single computer, it must buffer each message

Request size	ext3-async	xsyncfs
0–1 KB	0.064 (±0.025)	0.097 (±0.002)
1-10 KB	0.150 (±0.034)	0.180 (±0.001)
10-100 KB	1.084 (±0.052)	1.094 (±0.003)
100-1000 KB	10.253 (±0.098)	10.072 (±0.066)

The figure shows the mean time (in seconds) to request a file of a particular size during three trials of the SPECweb99 benchmark with 50 simultaneous connections. 90% confidence intervals are given in parentheses.

Figure 8. SPECweb99 latency results

until the file system data on which that message depends has been committed. In addition to the normal log data written by Apache, the SPECweb99 benchmark writes a log record to the file system as a result of each HTTP POST. Thus, small file writes are common during benchmark execution — a typical 45 minute run has approximately 150,000 file system transactions.

As shown in Figure 7, SPECweb99 throughput using xsyncfs is within 8% of the throughput achieved when ext3 is mounted asynchronously. In contrast to ext3, xsyncfs guarantees that the data associated with each POST request is durable before a client receives the POST response. The third bar in Figure 7 shows that SPECweb99 using ext3 mounted synchronously achieves 6% higher throughput than xsyncfs. Unlike the previous benchmarks, SPECweb99 writes little data to disk, so most writes are buffered by the drive cache. The last bar shows that xsyncfs achieves 7% better throughput than ext3 mounted synchronously with write barriers.

Figure 8 summarizes the average latency of individual HTTP requests during benchmark execution. On average, use of xsyncfs adds no more than 33 ms of delay to each request — this value is less than the commonly cited perception threshold of 50 ms for human users [5]. Thus, a user should perceive no difference in response time between xsyncfs and ext3 for HTTP requests.

4.7 Benefit of output-triggered commits

To measure the benefit of output-triggered commits, we also implemented an *eager commit* strategy for xsyncfs that triggers a commit whenever the file system is modified. The eager commit strategy still allows for group commit since multiple modifications are grouped into a single file system transaction while the previous transaction is committing. The next transaction will only start to commit once the commit of the previous transaction completes. The eager commit strategy attempts to minimize the latency of individual file system operations.

We executed the previous benchmarks using the eager commit strategy. Figure 9 compares results for the two strategies. The output-triggered commit strategy performs better than the eager commit strategy in every benchmark except SPECweb99, which creates so much output that the eager commit and output-triggered commit strategies perform very similarly. Since the eager commit strategy attempts to minimize the latency of a single operation, it sacrifices the opportunity to improve throughput. In contrast, the output-triggered commit strategy only minimizes latency after output has been generated that depends on a transaction; otherwise it maximizes throughput.

5 Related work

To the best of our knowledge, xsyncfs is the first local file system to provide high-performance synchronous I/O without requiring specialized hardware support or application modification. Further, xsyncfs is the first file system to use the causal relationship between file modifications and external output to decide when to commit data.

While xsyncfs takes a software-only approach to providing high-performance synchronous I/O, specialized hardware can achieve the same result. The Rio file cache [2] and the Conquest file system [31] use battery-backed main memory to make writes persistent. Durability is guaranteed only as long as the computer has power or the batteries remain charged.

Hitz et al. [7] store file system journal modifications on a battery-backed RAM drive cache, while writing file system data to disk. We expect that synchronous operations on Hitz's hybrid system would perform no better than ext3 mounted synchronously without write barriers in our experiments. Thus, xsyncfs could substantially improve the performance of such hybrid systems.

eNVy [33] is a file system that stores data on flash-based NVRAM. The designers of eNVy found that although reads from NVRAM were fast, writes were prohibitively slow. They used a battery-backed RAM write cache to achieve reasonable write performance. The write performance issues seen in eNVy are similar to those we experienced writing data to commodity hard drives. Therefore, it is likely that xsyncfs could also improve performance for flash file systems.

Xsyncfs's focus on providing both strong durability and reasonable performance contrasts sharply with the trend in commodity file systems toward relaxing durability to improve performance. Early file systems such as FFS [14] and the original UNIX file system [22] introduced the use of a main memory buffer cache to hold writes until they are asynchronously written to disk. Early file systems suffered from potential corruption when a computer lost power or an operating system crashed. Recovery often required a time consuming

Benchmark	Eager Commits	Output-Triggered Commits	Speedup
PostMark (seconds)	9.879 (±0.056)	8.668 (±0.478)	14%
Apache (seconds)	111.41 (±0.32)	109.42 (±0.71)	2%
MySQL 1 client (NOTPM)	3323 (±60)	4498 (±73)	35%
MySQL 20 clients (NOTPM)	3646 (±217)	4052 (±200)	11%
SPECweb99 (Kb/s)	312 (±1)	311(±2)	0%

This figure compares the performance of output-triggered commits with an eager commit strategy. Each result shows the mean of 5 trials, except SPECweb99, which is the mean of 3 trials. 90% confidence intervals are given in parentheses.

Figure 9. Benefit of output-triggered commits

examination of the entire state of the file system (e.g., running fsck). For this reason, file systems such as Cedar [6] and LFS [23] added the complexity of a write-ahead log to enable fast, consistent recovery of file system state. Yet, as was shown in our evaluation, journaling data to a write-ahead log is insufficient to prevent file system corruption if the drive cache reorders block writes. An alternative to write-ahead logging, Soft Updates [25], carefully orders disk writes to provide consistent recovery. Xsyncfs builds on this prior work since it writes data after returning control to the application and uses a write-ahead log. Thus, external synchrony could improve the performance of synchronous I/O with other journaling file systems such as JFS [1] or ReiserFS [16].

Fault tolerance researchers have long defined consistent recovery in terms of the output seen by the outside world [3, 11, 30]. For example, the *output commit* problem requires that, before a message is sent to the outside world, the state from which that message is sent must be preserved. In the same way, we argue that the guarantees provided by synchronous disk I/O should be defined by the output seen by the outside world, rather than by the results seen by local processes.

It is interesting to speculate why the principle of outside observability is widely known and used in fault tolerance research yet new to the domain of general purpose applications and I/O. We believe this dichotomy arises from the different scope and standard of recovery in the two domains. In fault tolerance research, the scope of recovery is the entire process; hence not using the principle of outside observability would require a synchronous disk I/O at every change in process state. In general purpose applications, the scope of recovery is only the I/O issued by the application (which can be viewed as an application-specific recovery protocol). Hence it is feasible (though still slow) to issue each I/O synchronously. In addition, the standard for recovery in fault tolerance research is well defined: a recovery system should lose no visible output. In contrast, the standard for recovery in general purpose systems is looser: asynchronous I/O is common, and even synchronous I/O is usually committed synchronously only to the volatile hard drive cache.

Our implementation of external synchrony draws upon two other techniques from the fault tolerance literature. First, buffering output until the commit is similar to deferring message sends until commit [12]. Second, tracking causal dependencies to identify what and when to commit is similar to causal tracking in message logging protocols [4]. We use these techniques in isolation to improve performance and maintain the appearance of synchronous I/O. We also use these techniques in combination via output-triggered commits, which automatically balance throughput and latency.

Transactions, provided by operating systems such as QuickSilver [24], TABS [28], and Locus [32], and by transactional file systems [10, 19], also give the strong durability and ordering guarantees that are provided by xsyncfs. In addition, transactions provide atomicity for a set of file system operations. However, transactional systems typically require that applications be modified to specify transaction boundaries. In contrast, use of xsyncfs requires no such modification.

6 Conclusion

It is challenging to develop simple and reliable software systems if the foundations upon which those systems are built are unreliable. Asynchronous I/O is a prime example of one such unreliable foundation. OS crashes and power failures can lead to loss of data, file system corruption, and out-of-order modifications. Nevertheless, current file systems present an asynchronous I/O interface by default because the performance penalty of synchronous I/O is assumed to be too large.

In this paper, we have proposed a new abstraction, external synchrony, that preserves the simplicity and reliability of a synchronous I/O interface, yet performs approximately as well as an asynchronous I/O interface. Based on these results, we believe that externally synchronous file systems such as xsyncfs can provide a better foundation for the construction of reliable software systems.

Acknowledgments

We thank Manish Anand, Evan Cooke, Anthony Nicholson, Dan Peek, Sushant Sinha, Ya-Yunn Su, our shepherd, Rob Pike, and the anonymous reviewers for feedback on this paper. The work has been supported by the National Science Foundation under award CNS-0509093. Jason Flinn is supported by NSF CAREER award CNS-0346686, and Ed Nightingale is supported by a Microsoft Research Student Fellowship. Intel Corp. has provided additional support. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, Microsoft, the University of Michigan, or the U.S. government.

References

- BEST, S. JFS overview. Tech. rep., IBM, http://www-128.ibm.com/developerworks/linux/library/l-jfs.html, 2000.
- [2] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RA-JAMANI, G., AND LOWELL, D. The Rio file cache: Surviving operating system crashes. In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (Cambridge, MA, October 1996), pp. 74–83.
- [3] ELNOZAHY, E. N., ALVISI, L., WANG, Y.-M., AND JOHN-SON, D. B. A survey of rollback-recovery protocols in messagepassing systems. ACM Computing Surveys 34, 3 (September 2002), 375–408.
- [4] ELNOZAHY, E. N., AND ZWAENEPOEL, W. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers C-*41, 5 (May 1992), 526–531.
- [5] FLAUTNER, K., AND MUDGE, T. Vertigo: Automatic performance-setting for Linux. In *Proceedings of the 5th Sympo*sium on *Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 105–116.
- [6] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, TX, 1987), pp. 155–162.
- [7] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994* USENIX Technical Conference (1994).
- [8] KATCHER, J. PostMark: A new file system benchmark. Tech. Rep. TR3022, Network Appliance, 1997.
- [9] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 7 (1978), 558–565.
- [10] LISKOV, B., AND RODRIGUES, R. Transactional file systems can be fast. In *Proceedings of the 11th SIGOPS European Workshop* (Leuven, Belgium, September 2004).
- [11] LOWELL, D. E., CHANDRA, S., AND CHEN, P. M. Exploring failure transparency and the limits of generic recovery. In Proceedings of the 4th Symposium on Operating Systems Design and Implementation (San Diego, CA, October 2000).
- [12] LOWELL, D. E., AND CHEN, P. M. Persistent messages in local transactions. In *Proceedings of the 1998 Symposium on Princi*ples of Distributed Computing (June 1998), pp. 219–226.
- [13] MCKUSICK, M. K. Disks from the perspective of a file system. ;login: 31, 3 (June 2006), 18–19.
- [14] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for unix. ACM Transactions on Computer Systems (TOCS) 2, 3 (August 1984), 181–197.
- [15] MYSQL AB. MySQL Reference Manual. http://dev.mysql.com/.
- [16] NAMESYS. ReiserFS. http://www.namesys.com/.

- [17] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.
- [18] OSDL. OSDL Database Test 2. http://www.osdl.org/.
- [19] PAXTON, W. H. A client-based transaction system to maintain data integrity. In *Proceedings of the 7th ACM Symposium on Op*erating Systems Principles (1979), pp. 18–23.
- [20] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 206–220.
- [21] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 235–248.
- [22] RITCHIE, D. M., AND THOMPSON, K. The UNIX time-sharing system. Communications of the ACM 17, 7 (1974), 365–375.
- [23] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. ACM Transactions on Computer Systems (TOCS) 10, 1 (February 1992), 26–52.
- [24] SCHMUCK, F., AND WYLIE, J. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 239–53.
- [25] SELTZER, M. I., GANGER, G. R., MCKUSICK, M. K., SMITH, K. A., SOULES, C. A. N., AND STEIN, C. A. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *USENIX Annual Technical Conference* (San Diego, CA, June 2000), pp. 18–23.
- [26] SILBERSCHATZ, A., AND GALVIN, P. B. Operating System Concepts (5th Edition). Addison Wesley, February 1998. p. 27.
- [27] SLASHDOT. Your Hard Drive Lies to You http://hardware.slashdot.org/article.pl?sid=05/05/13/0529252.
- [28] SPECTOR, A. Z., DANIELS, D., DUCHAMP, D., EPPINGER, J. L., AND PAUSCH, R. Distributed transactions for reliable systems. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA, December 1985), pp. 127– 146.
- [29] STANDARD PERFORMANCE EVALUATION CORPORATION. SPECweb99. http://www.spec.org/web99.
- [30] STROM, R. E., AND YEMINI, S. Optimistic Recovery in Distributed Systems. ACM Transactions on Computer Systems 3, 3 (August 1985), 204–226.
- [31] WANG, A.-I. A., REIHER, P., POPEK, G. J., AND KUENNING, G. H. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proceedings of the 2002 USENIX Annual Technical Conference* (Monterey, CA, June 2002).
- [32] WEINSTEIN, M. J., THOMAS W. PAGE, J., LIVEZEY, B. K., AND POPEK, G. J. Transactions and synchronization in a distributed operating system. In *Proceedings of the 10th ACM Sym*posium on Operating Systems Principles (Orcas Island, WA, December 1985), pp. 115–126.
- [33] WU, M., AND ZWAENEPOEL, W. eNVy: A non-volatile, main memory storage system. In Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (San Jose, CA, 1994), pp. 86– 97.

Type-Safe Disks

Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok Stony Brook University

Abstract

We present the notion of a *type-safe disk* (TSD). Unlike a traditional disk system, a TSD is aware of the pointer relationships between disk blocks that are imposed by higher layers such as the file system. A TSD utilizes this knowledge in two key ways. First, it enables active enforcement of invariants on data access based on the pointer relationships, resulting in better security and integrity. Second, it enables semantics-aware optimizations within the disk system. Through case studies, we demonstrate the benefits of TSDs and show that a TSD presents a simple yet effective general interface to build the next generation of storage systems.

1 Introduction

Pointers are the fundamental means by which modern file systems organize raw disk data into semantically-meaningful entities such as files and directories. Pointers define three things: (1) the semantic dependency between blocks (e.g., a data block is accessible only through a pointer from an inode block); (2) the logical grouping of blocks (e.g., blocks pointed to by the same indirect block are part of the same file or directory); and even (3) the importance of a block (e.g., blocks with many outgoing pointers are important because they impact the accessibility of a large set of blocks).

Despite the rich semantic information inherently available through pointers, pointers are completely opaque to disk systems today. Due to a narrow readwrite interface, storage systems view data simply as a raw sequence of uninterpreted blocks, thus losing all semantic structure imposed on the data by higher layers such as the file system or database system. This leads to the well-known *information gap* between the storage system and higher layers [8, 10]. Because of this information gap, storage systems are constrained in the range of functionality they can provide, despite the powerful processing capability and the great deal of low-level layout knowledge they have [25–27].

We propose the notion of a *type-safe disk* (TSD), a disk system that has knowledge of the pointer relationships between blocks. A TSD uses this knowledge in two key ways. First, semantic structure conveyed through pointers is used to enforce invariants on data access, providing better data integrity and security. For example, a TSD prevents access to an unallocated block.

Second, a TSD can perform various semantics-aware optimizations that are difficult to provide in the current storage hierarchy [25, 26].

A TSD extends the traditional block-based read-write interface with three new primitives: block allocation, pointer creation, and pointer removal. By performing block allocation and de-allocation, a TSD frees the file system from the need for free-space management. Similar in spirit to type-safe programming languages, a TSD also exploits its pointer awareness to perform automatic garbage collection of unused blocks; blocks which have no pointers pointing to them are reclaimed automatically, thus freeing file systems of the need to track reference counts for blocks in many cases.

We demonstrate the utility of a TSD through two prototype case studies. First, we show that a TSD can provide better data security by *constraining* data access to conform to implicit trust relationships conveyed through pointers. ACCESS (A Capability Conscious Extended Storage System) is a TSD prototype that provides an independent perimeter of security by constraining data access even when the operating system is compromised due to an attack. ACCESS enforces the invariant that for a block to be accessed, a parent block pointing to this block should have been accessed in the recent past.

ACCESS also allows certain top-level blocks to be associated with explicit read and write *capabilities* (i.e., per-block keys); access to all other blocks is then validated through the *implicit* capability vested by the fact that a parent block pointing to that block was successfully accessed before. Such *path-based capabilities* enable applications to encode arbitrary operation-level access policies and sharing modes by constructing separate pointer chains for different modes of access.

Our second case study is secure delete [13], a TSD prototype that automatically overwrites deleted blocks. When the last pointer to a block is removed, our secure deletion TSD schedules the block for overwrite and will not reuse it until the overwrite completes.

Overall, we find that a TSD presents an improved division of labor between file systems and storage. By building on the existing block-based interface, a TSD requires minimal modifications to the file system. All the modifications required are *implementation-level*, unlike *design-level* modifications that are required with brand new interfaces. To demonstrate the ease with which ex-

isting file systems can be ported to TSDs, we have modified two file systems, Linux Ext2 and VFAT, to use our TSD prototype; in both cases the changes were minimal.

Despite its simplicity, we find the interface to be quite powerful, since it captures the essence of a file system's semantic structure [24]. We describe how various kinds of functionality enhancements enabled by alternative approaches [19, 27] can be readily implemented in our model. We also find that the notion of type-safety is largely independent of the exact unit of block access. For example, even with recent proposals for an object-based interface to disks [11, 19], the ability to convey relationship between objects through pointers has benefits very similar to what we illustrate in our case studies.

We evaluate our prototype implementations by using micro-benchmarks and real workloads. We find that the primary performance cost in a TSD arises from the various forms of state that the disk tracks for block allocation, capability enforcement, etc. The costs, however, are quite minimal. For typical user workloads, a TSD has an overhead of just 3% compared to regular disks.

The rest of this paper is organized as follows. In Section 2 we discuss the utility of pointer information at the disk. Section 3 discusses the design and implementation of the basic TSD framework. In Section 4 we describe file system support for TSDs. Sections 6 and 7 present detailed case studies of two applications of TSDs: ACCESS and secure deletion. We evaluate all our prototype implementations in Section 8. We discuss related work in Section 9, and conclude in Section 10.

2 Motivation

In this section we present an extended motivation.

Pointers as a proxy for data semantics. The interlinkage between blocks conveys rich semantic information about the structure imposed on the data by higher layers. Most modern file systems and database systems make extensive use of pointers to organize disk blocks. For example, in a typical file system, directory blocks logically point to inode blocks which in turn point to indirect blocks and regular data blocks. Blocks pointed to by the same pointer block are often semantically related (e.g., they belong to the same file or directory). Pointers also define reachability: if an inode block is corrupt, the file system cannot access any of the data blocks it points to. Thus, pointers convey information about which blocks impact the availability of the file system to various degrees. Database systems are very similar in their usage of pointers. They have B-tree indexes that contain on-disk pointers, and their extent maps track the set of blocks belonging to a table or index.

In addition to being passively aware of pointer relationships, a type-safe disk takes it one step further. It

actively enforces invariants on data access based on the pointer knowledge it has. This feature of a TSD enables independent verification of file system operations; more specifically, it can provide an additional perimeter of security and integrity in the case of buggy file systems or a compromised OS. As we show in Section 6, a type-safe disk can limit the damage caused to stored data, even by an attacker with root privileges. We believe this active nature of control and enforcement possible with the pointer abstraction makes it powerful compared to other more passive information-based interfaces.

Pointers thus present a simple but general way of capturing application semantics. By aligning with the core abstraction used by higher-level application designs, a TSD has the potential to enable on-disk functionality that exploits data semantics. In the next subsection, we list a few examples of new functionality (some proposed in previous work in the context of alternative approaches) that TSDs enable.

Applications. There are several possible uses of TSDs. (1) Since TSDs are capable of differentiating data and pointers, they can identify metadata blocks as those blocks that contain outgoing pointers and replicate them to a higher degree, or distribute them evenly across all the disks. This could provide graceful degradation of availability as provided by D-GRAID [26]. (2) Using the knowledge of pointers, a TSD can co-locate blocks along with their reference blocks (blocks that point to them). In general, blocks will be accessed just after their pointer blocks are accessed, and hence there would be better locality during access. (3) TSDs can perform intelligent prefetching of data because of the pointer information. When a pointer block is accessed, a TSD can prefetch the data blocks pointed to by it, and store it in the on-disk buffers for improved read performance. (4) TSDs can provide new security properties using the pointer knowledge by enforcing implicit capabilities. We discuss this in detail in Section 6. (5) TSDs can perform automatic secure deletion of deleted blocks by tracking block liveness using pointer knowledge. We describe this in detail in Section 7.

3 Type-Safety at the Disk Level

Having pointer information inside the disk system enables enforcement of interesting constraints on data access. For example, a TSD allows access to only those blocks that are reachable through some pointer path. TSDs manage block allocations and enforce that every block must be allocated in the context of an existing pointer path, thus preventing allocated blocks from becoming unreachable. More interestingly TSDs enable disk-level enforcement of much richer constraints for data security as described in our case study in section 6.

Enforcing such access constraints based on pointer relationships between blocks is a restricted form of *type-safety*, a well-known concept in the field of programming languages. The type information that a TSD exploits, however, is narrower in scope: TSDs just differentiate between normal data and pointers.

We now detail the TSD interface, its operation, and our prototype implementation. Figure 1 shows the architectural differences between normal disks and a TSD.

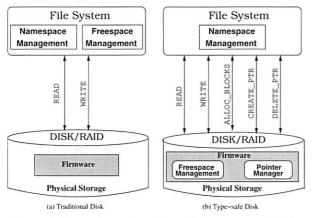


Figure 1: Comparison of traditional disks vs. type-safe disks

3.1 Disk API

A type-safe disk exports the following primitives, in addition to the basic block-based API:

- SET_BLOCKSIZE(Size): Sets the file system block size in bytes.
- ALLOC_BLOCKS(Ref, Hint, Count): Allocates Count number of new file system blocks from the disk-maintained free block list, and creates pointers to the allocated blocks, from block Ref. Allocated blocks need not be contiguous. Ref must be a valid block number that was previously allocated. Hint is the block number closest to which the new blocks should be allocated. Hint can be NULL, which means the disk can choose the new block totally at its own discretion. Returns an array of addresses of the newly allocated blocks, or NULL if there are not enough free blocks on the device.
- ALLOC_CONTIG_BLOCKS(Ref, Hint, Count):
 Follows the same semantics as ALLOC_BLOCKS, except that it allocates Count number of contiguous blocks if available.
- CREATE_PTR(Src, Dest): Creates a pointer from block Src to block Dest. Both Src and Dest must be previously allocated. Returns success or failure.
- DELETE_PTR(Src, Dest): Deletes a pointer from block Src that points to block Dest. Semantics similar to CREATE_PTR.
- GET_FREE: Returns the number of free blocks left.

3.2 Managing Block Pointers

A TSD needs to maintain internal data-structures to keep track of all pointers between blocks. It maintains a pointer tracking table called PTABLE that stores the set of all pointers. The PTABLE is indexed by the source block number and each table entry contains the list of destination block numbers. A new PTABLE entry is added every time a pointer is created. Based on pointer information, TSD disk blocks are classified into three kinds: (a) Reference blocks: blocks with both incoming and outgoing pointers (such as inode blocks). (b) Data blocks: blocks without any outgoing pointers but just incoming pointers. (c) Root blocks: a pre-determined set of blocks that contain just outgoing pointers but not incoming pointers. Root blocks are never allocated or freed, and they are statically determined by the disk. Root blocks are used for storing boot information or the primary metadata block of file systems (e.g., the Ext2 super block).

3.3 Free-Space Management

To perform free-space management at the disk level, we track live and free blocks. A TSD internally maintains an allocation bitmap, ALLOC-BITMAP, containing one bit for every logical unit of data maintained by the higher level software (e.g., a file system block). The size of a logical unit is set by the upper-level software through the SET_BLOCKSIZE disk primitive. When a new block need to be allocated, the TSD can choose a free block closest to the hint block number passed by the caller. Since the TSD can exploit the low level knowledge it has, it chooses a block number which requires the least access time from the hint block.

TSDs use the knowledge of block liveness (a block is defined to be dead if it has no incoming pointers) to perform garbage collection. Unlike traditional garbage collection systems in programming languages, garbage collection in TSD happens synchronously during a particular DELETE_PTR call which deletes the last incoming pointer to a block. A TSD maintains a reference count table, RTABLE, to speed up garbage collection. The reference count of a block gets incremented every time a new incoming pointer is created and is decremented during pointer deletions. When the reference count of a block drops to zero during a DELETE_PTR call, the block is marked free immediately. A TSD performs garbage collection one block at a time as apposed to performing cascading deletes. Garbage collection of reference blocks with outgoing pointers is prevented by disallowing deletion of the last pointer to a reference block before all outgoing pointers in it are deleted.

3.4 Consistency

As TSDs maintain separate pointer information, TSD pointers could become inconsistent with the file system

pointers during system crashes. Therefore, upon a system crash, the consistency mechanism of the file system is triggered which checks file system pointers against TSD pointers and first fixes any inconsistencies between both. It then performs a regular scan of the file system to fix file system inconsistencies and update the TSD pointers appropriately. For example, if the consistency mechanism creates a new inode pointer to fix an inconsistency, it also calls the CREATE_PTR primitive to update the TSD internal pointers. Alternatively, we can obviate the need for consistency mechanisms by just modifying file systems to use TSD pointers instead of maintaining their own copy in their meta-data. However, this involves wide-scale modifications to the file system.

File system integrity checkers such as fsck for TSDs have to run in a privileged mode so that they can perform a scan of the disk without being subjected to the constraints enforced by TSDs. This privileged mode can use a special administrative interface that overrides TSD constraints and provides direct access to the TSD pointer management data-structures.

Block corruption. When a block containing TSD-maintained pointer data-structures gets corrupted the pointer information has to be recovered, as the data blocks pertaining to the pointers could still be reachable through the file system meta-data. Block corruption can be detected using well-known methods such as check-summing. Upon detection, the TSD notifies the file system, which recreates the lost pointers from its meta-data.

3.5 Prototype Implementation

We implemented a prototype TSD as a pseudo-device driver in Linux kernel 2.6.15 that stacks on top of an existing disk block driver. It contains 3,108 lines of kernel code. The TSD layer receives all block requests, and redirects the common read and write requests to the lower level device driver. The additional primitives required for operations such as block allocation and pointer management are implemented as driver ioctls.

We implemented PTABLE and RTABLE as in-memory hash tables which gets written out to disk at regular intervals of time through an asynchronous commit thread. In implementing the RTABLE, we add an optimization to reduce the number of entries maintained in the hash table. We add only those blocks whose reference count is greater than one. A block which is allocated and which does not have an entry in the RTABLE is deemed to have a reference count of one and an unallocated block (as indicated by the ALLOC_BITMAP) is deemed to have a reference count of zero. This significantly reduces the size of our RTABLE, because most disk blocks have reference counts of zero or one (e.g., all data blocks have reference counts zero or one).

4 File System Support

We now describe how a file system needs to be modified to use a TSD. We first describe the general modifications required to make any file system work with a TSD. Next, we describe our modifications to two file systems, Linux Ext2 and VFAT, to use our framework.

Since TSDs perform free-space management at the disk-level, file systems using TSD are freed from the complexity of allocation algorithms, and tracking free block bitmaps and other related meta-data. However, file systems now need to call the disk API to perform allocations, pointer management, and getting the free blocks count. The following are the general modifications required to existing file systems to support type-safe disks:

- The mkfs program should set the file system block size using the SET_BLOCKSIZE primitive, and store the primary meta-data block of the file system (e.g., the Ext2 super block) in one of the TSD root blocks. Note that the TSD root blocks are a designated set of well-known blocks known to the file system.
- The free-space management sub-system should be eliminated from the file system, and TSD API should be used for block allocations. The file system routine that estimates free-space, should call the GET_FREE disk API, instead of consulting its own allocation structures.
- Whenever file systems add new pointers to their meta-data, CREATE_PTR disk primitive should be called to create a TSD pointer. Similarly, the DELETE_PTR primitive has to be called when pointers are removed from the file system.

In the next two sub-sections we describe the modifications that we made to the Ext2 and the VFAT file systems under Linux, to support type-safe disks.

4.1 Ext2TSD

We modified the Linux Ext2 file system to support typesafe disks; we call the modified file system *Ext2TSD*. The Ext2 file system groups together a fixed number of sequential blocks into a block group and the file system is managed as a series of block groups. This is done to keep related blocks together. Each block group contains a copy of the super block, inode and block allocation data-structures, and the inode blocks. The inode table is a contiguous array of blocks in the block group that contain on-disk inodes.

To modify Ext2 to support TSDs, we removed the notion of block groups from Ext2. Since allocations and de-allocations are done by using the disk API, the file system need not group blocks based on their order. However, to perform easy inode allocation in tune with Ext2, we maintain inode groups which we call ISEGMENTS. Each isegment contains a segment descriptor that has an

inode bitmap to track the number of free inodes in that isegment. The inode allocation algorithm of Ext2TSD is same as that of Ext2. The mkfs user program of Ext2TSD writes the super block, and allocates the inode segment descriptor blocks, and inode tables using the allocation API of the disk. It also creates pointers from the super block to all blocks containing isegment descriptors and inodes tables.

The organization of file data in Ext2TSD follows the same structure as Ext2. When a new file data or indirect block is allocated, Ext2TSD calls ALLOC_BLOCKS with the corresponding inode block or the indirect block as the reference block. While truncating a file, Ext2TSD just deletes the pointers in the indirect block branches in the right order such that all outgoing pointers from the parent block to its child blocks are deleted before deleting the incoming pointer to the parent block. Thus blocks belonging to truncated or deleted files are automatically reclaimed by the disk.

In the Ext2 file system, each directory entry contains the inode number for the corresponding file or directory. This is a logical pointer relationship between the directory block and the inode block. In our implementation of Ext2TSD, we create physical pointers between a directory block and the inode blocks corresponding to the inode numbers contained in every directory entry in the directory block. Modifying the Ext2 file system to support TSD was relatively simple. It took 8 days for us to build Ext2TSD starting from a vanilla Ext2 file system. We removed 538 lines of code from Ext2 which are mostly the code required for block allocation and bitmap management. We added 90 lines of new kernel code and modified 836 lines of existing code.

4.2 VFATTSD

The next file system we consider is VFAT, a file system with origins in Windows. Specifically, we consider the Linux implementation of VFAT. We chose to modify VFAT to support TSDs because it is sufficiently different in architecture from Ext2 and hence shows the generality of the pointer level abstraction provided by TSDs. We call our modified file system *VFATTSD*.

The VFAT file system contains an on-disk structure called the File Allocation Table (FAT). The FAT is a contiguous set of blocks in which each entry contains the logical block number of the next block of a file or a directory. To get the next block number of a file, the file system consults the FAT entries corresponding to the previous block of the file. Each file or directory's first block is stored as part of the directory entry in the corresponding directory block. The FAT entry corresponding to the last block of a file contains an EOF marker. VFAT tracks free blocks by having a special marker in the FAT entry corresponding to the blocks.

In the context of TSDs, we need not use the FAT to track free blocks. All block allocations are done using the allocation API provided by a TSD. The mkfs file system creation program allocates and writes the FAT blocks using the disk API. Modifying the VFAT file system to support TSDs was substantially simpler compared to Ext2, as VFAT does not manage data blocks hierarchically. We had to maintain substantially lesser number of pointers.

In VFAT, we created pointers from each directory block to all blocks belonging to files which have their directory entries in the directory block. Each FAT block points to the block numbers contained in the entries present within. The TSD therefore tracks all blocks belonging to files in the same directory block. Also, all the directory blocks and the FAT blocks contain outgoing pointers. The disk can track the set of all metadata blocks present in the file system by just checking if a block is a data block or a reference block.

Modifying the VFAT file system to support TSD was relatively straightforward. It took 4 days for us to build VFATTSD from the VFAT file system. We added 83 lines of code, modified 26 lines of code, and deleted 71 lines of code. The deleted code belonged to the free space management component of VFAT.

5 Other Usage Scenarios

In this section we discuss how the TSD abstraction that we have presented fits three other usage scenarios.

RAID systems. The TSD architecture requires a one-to-one correspondence between a file system and a TSD. However, in aggregated storage architectures such as RAID, a software or hardware layer could exist between file systems and TSDs. In this scenario, the file system gets distributed across several TSDs and no single piece has all the pointer information.

To realize the benefits of TSDs in this model, we propose the following solution: all the layers between the file systems and the TSDs should be aware of the TSD interface. Reference blocks should be replicated across the TSDs, and the software layer that performs aggregation should route the pointer management calls to the appropriate TSD that contains the corresponding data blocks. Therefore, in an aggregated storage system, each TSD contains a copy of all the reference blocks, but has only a subset of pointers pertaining to the data blocks in them. This ensures that whatever information that a single TSD has is sufficient for its own internal operations such as garbage collection, and the global structure can be used by the aggregation layer by combining the information present in each of the TSDs. The aggregation layer intercepts the CREATE_PTR and DELETE_PTR calls and invokes the disk primitives of the TSD which contains the corresponding data blocks. The aggregation layer also contains an allocation algorithm to route the ALLOC_BLOCKS call from the file system to the appropriate TSD. For example, if there are three TSDs in a software RAID system and a file is striped across the three, all the disks will contain the file's inode block. However, each disk's pointer data-structures will contain only the pointers from the inode blocks to those data blocks that are present in that disk. In this case, the first disk only contains pointers from the inode block to block offsets 0, 3, 6, and so on. A related idea explored in the context of Chunkfs is allowing files and directories to span across different file systems by having *continuation inodes* in each file system [14].

Journalling file systems. Journalling file systems maintain a persistent log of operations for easy recovery after a crash. A journalling file system that uses a TSD should pre-allocate the journal blocks using the ALLOC_BLOCKS primitive with the reference block as one of the root blocks. For example, Ext3 can create pointers from the super block to all the journal blocks. Journalling file systems should also update TSD pointers during journal recovery, using the pointer management API of the TSD, to ensure that the TSD pointer information is in sync with the file system meta-data.

Software dependent on physical locations. Software that needs to place data in the exact physical location on the disk, such as some physical backup tools, may not benefit as much from the advantages of TSDs. This is because TSDs do not provide explicit control to the upper level software to choose the precise location of a block to allocate. However, such software can be supported by TSDs by using common techniques such as preallocating all blocks in the disk and then managing them at the software level. For example, a log-structured file system can allocate all TSD blocks using the ALLOC_BLOCKS primitive during bootstrapping, and then perform its normal operation within that range of blocks.

6 Case Study: ACCESS

We describe how type-safety can enable a disk to provide better security properties than existing storage systems. We designed and implemented a secure storage system called ACCESS (A Capability Conscious Extended Storage System) using the TSD framework; we then built a file system on top, called Ext2ACCESS.

Protecting data confidentiality and integrity during intrusions is crucial: attackers should not be able to read or write on-disk data even if they gain root privileges. One solution is to use encryption [6, 30]; this ensures that intruders cannot decipher the data they steal. However, encryption does not protect the data from being overwritten or destroyed. An alternative is to use explicit

disk-level *capabilities* to control access to data [1, 11]. By enforcing capabilities independently, a disk enables an additional perimeter of security even if the OS is compromised. Others explored using disk-level versioning that never overwrites blocks, thus enabling the recovery of pre-attack data [28].

ACCESS is a type-safe disk that uses pointer information to enforce *implicit path-based* capabilities, obviating the need to maintain explicit capabilities for all blocks, yet providing similar guarantees.

ACCESS has five design goals. (1) Provide an infrastructure to limit the scope of confidentiality breaches on data stored on local disks even when the attacker has root privileges or the OS and file systems are compromised. (2) The infrastructure should also enable protection of stored data against damage even in the event of a network intruder gaining access to the raw disk interface. (3) Support efficient and easy revocation of authentication keys, which should not require costly re-encryptions upon revocation. (4) Enable applications to use the infrastructure to build strong and easy-to-use security features. (5) Support data recovery through administrative interfaces even when authentication tokens are lost.

6.1 Design

The primitive unit of storage in today's commodity disks is a fixed-size disk block. Authenticating every block access using a capability is too costly in terms of performance and usability. Therefore, there needs to be some criteria by which blocks are grouped and authenticated together. Since TSDs can differentiate between normal data and pointers, they can perform logical grouping of blocks based on the reference blocks pointing to them. For example, in Ext2 all data blocks pointed to by the same indirect block belong to the same file.

ACCESS provides the following guarantee: a block x cannot be accessed unless a valid reference block y that points to this block x is accessed. This guarantee implies that protecting access to data simply translates to protecting access to the reference blocks. Such grouping is also consistent with the fact that users often arrange files of related importance into individual folders. Therefore, in ACCESS, a single capability would be sufficient to protect a logical working set of user files. Reducing the number of capabilities required is not only more efficient, but also more convenient for users.

In ACCESS, blocks can have two capability strings: a read and a write capability (we call these *explicit capabilities*). Blocks with associated explicit capabilities, which we call *protected* blocks, can be read or written only by providing the appropriate capability. By performing an operation on a block Ref using a valid capability, the user gets an *implicit capability* to perform the same operation on all blocks pointed to by Ref,

which are not directly protected (capability inheritance). If a particular reference block i points to another block j with associated explicit capabilities, then the implicit capability of i is not sufficient to access j; the explicit capability of j is needed to perform operations on it.

As all data and reference blocks are accessed using valid pointers stored on disk, root blocks are used to bootstrap the operations. In ACCESS, there are two kinds of access modes: (1) All protected blocks are accessed by providing the appropriate capability for the operation. (2) Blocks which are not protected can inherit their capability from an authenticated parent block.

ACCESS meta-data. ACCESS maintains a table named KTABLE indexed by the block number, to store the blocks' read and write capabilities. It also maintains a temporal access table called LTABLE which is indexed by the reference block number. The LTABLE has entries for all reference blocks whose associated implicit capabilities have not timed out. The timed out entries in the LTABLE are periodically purged.

Preventing replay attacks. In ACCESS, data needs to be protected even in situations where the OS is compromised. Passing clear-text capabilities through the OS interface could lead to replay attacks by a silent intruder who eavesdrops capabilities. To protect against this, ACCESS associates a sequence number with capability tokens. To read a protected block, the user has to provide a HMAC checksum of the capability (C_u) concatenated with a sequence number (S_u) $(H_u =$ $HMAC(C_u + S_u, C_u)$). This can be generated using an external key card or a hand-held device that shares sequence numbers with the ACCESS disk system. Each user has one of these external devices, and ACCESS tracks sequence numbers for each user's external device. Upon receiving H_u for a block, ACCESS retrieves the capability token for that block from the KTABLE and computes $H_A = HMAC(C_A + S_A, C_A)$, where C_A and S_A are the capability and sequence number for the block, and are maintained by ACCESS. If H_u and H_A do not match, ACCESS denies access. Skews in sequence numbers are handled by allowing a window of valid sequence numbers at any given time.

ACCESS operation. During every reference block access, an optional timeout interval (Interval) can be provided, during which the implicit capabilities associated with that reference block will be active. Whenever a reference block Ref is accessed successfully, an LTABLE entry is added for it. This entry stays until Interval expires. It is during this period of time, that we call the temporal window, all child blocks of Ref which are not protected inherit the implicit capability of accessing Ref. Once the timeout interval expires, all further accesses to the child blocks are denied. This condition

should be captured by the upper level software, which should prompt the user for the capability token, and then call the disk primitive to renew the timeout interval for Ref. The value of Interval can be set based on the security and convenience requirements. Long-running applications that are not interactive in nature should choose larger timeout intervals.

At any instant of time when the OS is compromised, the subset of blocks whose temporal window is active will be vulnerable to attack. This subset would be a small fraction of the entire disk data. The amount of data vulnerable during OS compromises can be reduced by choosing short timeout intervals. One can also force the timeout of the temporal window using the FORCE_TIMEOUT disk primitive described below.

ACCESS API. To design the ACCESS API, we extended the TSD API (Section 3) with capabilities, and added new primitives for managing capabilities and timeouts. Note that some of the primitives described below let the file system specify the reference block through which the implicit capability chain is established. However, as we describe later, this is only used as a hint by the disk system for performance reasons; ACCESS maintains its own structures that validate whether the specified reference block was indeed accessed, and it has a pointer to the actual block being accessed. In this section when we refer to read or write *capabilities*, we mean the HMAC of the corresponding capabilities and a sequence number.

- SET_CAPLEN(Length): Sets the length of capability tokens. This setting is global.
- ALLOC_BLOCKS(Ref, Ref_r|C_w, Count): Operates similar to the TSD ALLOC_BLOCKS primitive with the following two changes. (1) If Ref is protected the call takes the write capability of Ref, C_w; (2) otherwise, the call takes the reference block Ref_r of Ref, to verify that the caller has write access to Ref.
- ALLOC_CONTIG_BLOCKS(Ref, Ref_r|C_w, Count):
 Same as the ALLOC_BLOCKS primitive, but allocates contiguous blocks.
- READ(Bno, Ref|C_{rw}, Timeout): Reads the block represented by Bno. Ref is the reference block that has a pointer to Bno. C_{rw} is either the read or the write capability of block Bno. The second argument of this primitive must be Ref if Bno is not protected for read, and must be C_{rw} if Bno is protected. Timeout is the timeout interval.
- WRITE($Bno, Ref|C_w, timeout$): Writes the block represented by Bno. C_w is the write capability of Bno. Other semantics are similar to READ.
- CREATE_PTR($Src, Dest, Ref_s | C_{sw}, C_{dw} | Ref_{dw}$): Creates a pointer from block Src to block Dest. If

Src or Dest are protected, their capabilities have to be provided. For blocks which are not protected, the caller must provide valid reference blocks which point to Src and Dest. Note that although the pointer is created only from the source block, we need the write capability for the destination block as well; without this requirement, one can create a pointer to any arbitrary block and gain implicit write capabilities on that block.

- DELETE_PTR($Src, Dest, Ref_s | C_{sw}$): Deletes a pointer from block Src to block Dest. Write credentials for Src has to be provided.
- KEY_CONTROL $(Bno, C_{ow}, C_{nr}, C_{nw}, Ref)$: This sets, unsets, or changes the read and write capabilities associated with the block Bno. C_{ow} is the old write capability of Bno. C_{nr} and C_{nw} are the new read and write capabilities respectively. A reference block Ref that has a pointer to Bno needs to be passed only while setting the write key for a block that did not have a write capability before. For all other operations, like unsetting keys or changing keys, Ref need not be specified because C_{ow} can be used for authentication.
- RENEW_CAPABILITY(Ref, C_{rw}, Interval): Renews the capability for a given reference block.
 C_{rw} is the read or write key associated with Ref. Interval is the timeout interval for the renewal.
- FORCE_TIMEOUT(Ref): Times out the implicit capabilities associated with reference block Ref.
- SET_BLOCKSIZE and GET_FREE TSD primitives (Section 3) can be called through the secure administrative interface discussed in Section 6.3.

6.2 Path-Based Capabilities

Capability systems often use capabilities at the granularity of *objects* (e.g., physical disk blocks, or memory pages); each object is associated with a capability that needs to be presented to gain access.

In contrast, the implicit capabilities used by ACCESS are path-level. In other words, they authenticate an access based on the path through which the access was made. This mechanism of authenticating paths instead of individual objects is quite powerful in enabling applications to encode arbitrary trust relationships in those paths. For example, a database system could have a policy of allowing any user to access a specific row in a table by doing an index lookup of a 64-bit key, but restrict scans of the entire table only to privileged users. With per-block (or per-row) capabilities, this policy cannot be enforced at the disk unless the disk is aware of the scan and index lookup operations. With path-based capabilities, the database system could simply encode this policy by constructing two separate pointer chains: one going from each block in the table to the next, and another from the index block to the corresponding table block—and just have different keys for the start of both these chains. Thus, the same on-disk data item can be differentiated for different *application-level* operations, while the disk is oblivious to these operations.

Another benefit of the path-based capability abstraction is that it enables richer modes of sharing in a file system context. Let's assume there are n users in a file system and each user shares a subset of files with another user. With traditional encryption or per-object capability systems, users has to use a separate key for each other user that shares their files; this is clearly a key management nightmare (with arbitrary sharing, we would need n^2 keys). In our model, users can use the same key regardless of how many users share pieces of their data. To enable another user to share a file, all that needs to be done is a separate link be created from the other user's directory to this specific file. The link operation needs to take capabilities of both users, but once the operation is complete, the very fact that the pointer linkage exists will enable the sharing, but at the same time limit the sharing to only those pieces of data explicitly shared.

6.3 Key Revocation and Data Recovery

ACCESS enables efficient and easy key revocation. In normal encryption based security systems, key revocation could become pretty costly in proportion to the size of the data, as all data have to be decrypted and reencrypted with the new key. With ACCESS, one just changes the capability for the reference blocks instead of the entire set of data blocks. Data need not be modified at all while revoking capabilities.

Secure key backup is a major task in any encryptionbased data protection system. Once an encryption key is lost, usually the data is fully lost and cannot be recovered. ACCESS does not have this major problem. Data is not encrypted at all, and hence even if keys are lost, data can be retrieved or the keys may be reset using the administrative interface described below.

Often system administrators need to perform backup and or administrative operations for which the restricted ACCESS interface might not be sufficient. ACCESS will have a secure administrative interface, which could be through a special hardware port requiring physical access, in combination with a master key. Using the secure administrative interface, the administrator can backup files, delete unimportant files, etc., because the data is not stored internally in encrypted format.

6.4 ACCESS Prototype

We extended our TSD prototype to implement ACCESS. We implemented additional hash tables for storing the KTABLE and LTABLE required for tracking capabilities and temporal access locality. All in-memory hash tables

were periodically committed to disk through an asynchronous commit thread. The allocation and pointer management ioctls in TSD were modified to take capabilities or reference blocks as additional arguments. We implemented the KEY_CONTROL primitive as a new ioctl in our pseudo-device driver.

To authenticate the read and write operations, we implemented a new ioctl, KEY_INPUT. We did this to simplify our implementation and not modify the generic block driver. The KEY_INPUT ioct1 takes the block number and the capabilities (or reference blocks) as arguments. The upper level software should call this ioctl before every read or write operation to authenticate the access. Internally, the disk validates the credentials provided during the ioctl and stores the success or failure state of the authentication. When a read or write request is received, ACCESS checks the state of the previous KEY_INPUT for the particular block to allow or disallow access. Once access is allowed for an operation, the success state is reset. When a valid KEY_INPUT is not followed by a subsequent read or write for the block (e.g., due to software bugs), we time out the success state after a certain time interval.

6.5 The Ext2ACCESS File System

We modified the Ext2TSD file system described in Section 4.1 to support ACCESS; we call the new file system *Ext2ACCESS*. To demonstrate a usage model of ACCESS disks, we protected only the inode blocks of Ext2ACCESS with read and write capabilities. All other data blocks and indirect blocks had implicit capabilities inherited from their inode blocks. This way users can have a single read or write capability for accessing a whole file. An alternative approach may be to protect only directory inode blocks. ACCESS provides an infrastructure for implementing security at different levels, which upper level software can use as needed.

To implement per-file capabilities, we modified the Ext2 inode allocation algorithm. Ext2 stores several inodes in a single block; so in Ext2ACCESS we needed to ensure that an inode block has only those inodes that share the same capabilities. To handle this, we associated a *capability table* with every isegment (Section 4.1). The capability table persistently stores the checksums of the capabilities of every inode block in the particular isegment. Whenever a new inode needs to be allocated, an isegment with the same key is chosen if available.

Ext2ACCESS has two file system ioct1s, called SET_KEY and UNSET_KEY, which can be used by user processes to set and unset capabilities for files. The life of a user's key in kernel memory can be decided by the user. For example, a user can call the SET_KEY ioct1 before an operation and then immediately call the UNSET_KEY ioct1 after the operation is completed to

erase the capability from kernel memory; in this case the life of the key in kernel memory is limited to a single operation. Ext2ACCESS uses the KEY_INPUT device ioctl of ACCESS to send the user's key before reading an inode block. For all other blocks, it sends the corresponding reference block as an implicit capability, for temporal authentication.

An issue that arises in Ext2ACCESS is that general file system meta-data such as super block and descriptors need to be written to all the time (and hence must have their capabilities in memory). This can potentially make them vulnerable to modifications by attackers. We address this vulnerability by mapping these blocks to root blocks and enforce that no pointer creations or deletions can be made to root blocks except through an administrative interface. Accordingly, mkfs creates set of pointers to the relevant inode bitmap and isegment descriptor blocks, but this cannot change after that. Thus, we ensure confidentiality and write protection of all protected user files and directories.

Although the above solution protects user data during attacks, the contents of the metadata blocks themselves could be modified (for example, free block count, inode allocation status, etc). Although most of this information can be reconstructed by querying the pointer structure from the disk, certain pieces of information are hard to reconstruct. Our current implementation does not handle this scenario, but there are various solutions to this problem. First, we could impose that the disk perform periodic snapshotting of root blocks; since these are very few in number, the overhead of snapshotting will be minimal. Alternatively, some amount of NVRAM could be used to buffer writes to these global metadata blocks and periodically (say once a day) an administrator "commits" these blocks to disk using a special capability after verifying its integrity.

7 Case Study: Secure Deletion

In this section we describe our next case study: a disk system that automatically performs secure deletion of blocks that are freed. We begin with a brief motivation and then move on to the design and implementation of our *Secure Deletion Type-Safe Disk* (SDTSD).

Data security often includes the ability to delete data such that it cannot be recovered [3, 13, 23]. Several software-level mechanisms exist today that delete disk data securely [16, 22]. However, these mechanisms are fundamentally insecure compared to disk-level mechanisms [25], because the former do not have knowledge of disk internals and therefore cannot guarantee that deleted data is overwritten.

Since a TSD automatically tracks blocks that are not used, obtaining liveness information about blocks is simple as described in Section 3. Whenever a block is

garbage collected, an SDTSD just needs to securely delete the block by overwriting it one or more times. The SDTSD must also ensure that a garbage collected block that is not yet securely deleted is not re-allocated; an SDTSD achieves this by deferring the update of the ALLOC_BITMAP until a block is securely deleted.

To improve performance, an SDTSD overwrites blocks in batches. Blocks that are garbage collected are automatically added to a secure-deletion list. This list is periodically flushed and the blocks to be securely deleted are sorted for sequential access. Once a batch of blocks is overwritten multiple times, the ALLOC_BITMAP is updated to mark all those blocks as free.

We extended our prototype TSD framework described in Section 3.5 to implement secure-deletion functionality. Whenever a block is garbage collected, we add the block number to a list. An asynchronous kernel thread wakes up every second to flush the list into a buffer, sort it, and perform overwrites. The number of overwrites per block is configurable. We added 403 lines of kernel code to our existing TSD prototype.

8 Evaluation

We evaluated the performance of our prototype TSD framework in the context of Ext2TSD and VFATTSD. We also evaluated our prototype implementations of ACCESS and secure delete. We ran general-purpose workloads and also micro-benchmarks on our prototypes and compared them with unmodified Ext2 and VFAT file systems on a regular disk. This section is organized as follows: first we talk about our test platform, configurations, and procedures. Next, we analyse the performance of the TSD framework with the Ext2TSD and VFATTSD file systems. Finally, we evaluate our prototypes for ACCESS and SDTSD.

Test infrastructure. We conducted all tests on a 2.8GHz Xeon with 1GB RAM, and a 74GB, 10Krpm, Ultra-320 SCSI disk. We used Fedora Core 4, running a vanilla Linux 2.6.15 kernel. To ensure a cold cache, we unmounted all involved file systems between each test. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student-t distribution. In each case, the half-widths of the intervals were less than 5% of the mean. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it. We recorded disk statistics from /proc/diskstats for our test disk. We provide the following detailed disk-usage statistics: the number of read I/O requests (rio), number of write I/O requests (wio), number of sectors read (rsect), number of sectors written (wsect), number of read requests merged (rmerge), number of write requests merged (wmerge),

total time taken for read requests (ruse), and the total time taken for write requests (wuse).

Benchmarks and configurations. We used Postmark v1.5 to generate an I/O-intensive workload. Postmark stresses the file system by performing a series of operations such as directory lookups, creations, and deletions on small files [17]. For all runs, we ran Postmark with 50,000 files and 250,000 transactions.

To simulate a relatively CPU-intensive user workload, we compiled the Linux kernel source code. We used a vanilla Linux 2.6.15 kernel, and analyzed the overheads of Ext2TSD and Ext2ACCESS, for the untar, make oldconfig, and make operations combined.

To isolate the overheads of individual file system operations, we ran micro-benchmarks that analyze the overheads associated with the create, lookup, and unlink operations. For all micro-benchmarks, we used a custom user program that creates 250 directories and 1,000 files in each of these directories, creating a total of 250,000 files. For performing lookups, we called the stat operation on each of these files. We called stat by specifying the full path name of the files so that readdir was not called. For the unlink micro-benchmarks, we removed all 250,000 files created.

Unless otherwise mentioned, the system time overheads were caused by the hash table lookups required during the CREATE_PTR and DELETE_PTR TSD calls. This CPU overhead is due to the fact that our prototype is implemented as a pseudo-device driver that runs on the same CPU as the file system. In a real TSD setting, the hash table lookups will be performed by the processor embedded in the disk and hence will not influence the overheads on the host system.

8.1 Ext2TSD

We analyze the overheads of Ext2TSD over our TSD framework in comparison with the overheads of regular Ext2 over a regular disk. We discuss the Postmark, kernel compilation, and micro-benchmark results.

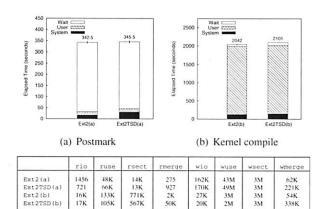


Figure 2: Postmark and Kernel compile results for Ext2TSD

Postmark. Figure 2(a) shows the comparison of Ext2TSD over TSD with regular Ext2. Ext2TSD has a system time overhead of 81% compared to regular Ext2. This is because of the hash table lookups required for creating and deleting pointers. The wait time of the Ext2TSD configurations was 5% lower than regular Ext2. This is because of better spatial locality in Ext2TSD for the Postmark workload. This is evidenced by the higher rmerge and wmerge values of Ext2TSD compared to Ext2. Ext2's allocation policy takes into account future file growth and hence leaves free blocks between newly created files. In Ext2TSD, we did not implement this policy and hence we have better locality for small files. Overall, the elapsed times for Ext2 and Ext2TSD under Postmark are similar.

Kernel compile. The Ext2TSD results for the kernel compilation benchmark are shown in Figure 2(b). The wait time overhead for Ext2TSD is 77%. This increase in wait time is not because of increase in I/O, as shown in the disk statistics. The increase is because of the increased sleep time of the Postmark process context while the TSD commit thread (described in Section 3.5) preempts it to commit the hash tables. The asynchronous commit thread runs in a different context and has to traverse all hash tables to commit them, taking more system time, which is reflected as wait time in the context of Postmark. Since a kernel compile is not an I/O-intensive workload, the system time overhead is lower than the overhead for Postmark. The elapsed time overhead of Ext2TSD compared to Ext2 under this benchmark is 3%.

Micro-benchmarks. We ran the CREATE, LOOKUP, and UNLINK micro-benchmarks on Ext2TSD and compared them with Ext2TSD.

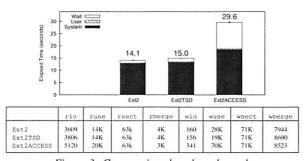


Figure 3: Create micro-benchmark results

Figure 3 shows the results for the CREATE microbenchmark. The wait time overhead is 36%, which is due to the increase in the sleep time due to CPU context switches required for the TSD commit thread. Since this benchmark has a significant system time component, this is more pronounced. The wuse and ruse values in the disk statistics have not increased and hence the higher wait time is not because of additional I/O.

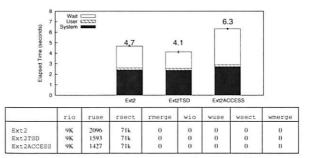


Figure 4: Lookup micro-benchmark results

Figure 4 shows the results of the LOOKUP microbenchmark. Ext2TSD had a 12% lower elapsed time than Ext2. This is mainly because of the 24% savings in wait time thanks to the better spatial locality evidenced by the ruse value of the disk statistics.

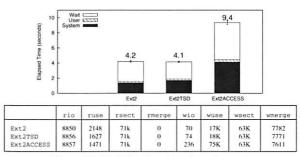


Figure 5: Unlink micro-benchmark results

Figure 5 shows the results of the UNLINK microbenchmark. Ext2TSD is comparable to Ext2 in terms to elapsed time. This is because the 21% increase in system time is compensated for by the 12% decrease in the wait time, due to better spatial locality.

8.2 VFATTSD

We evaluated the overheads of VFATTSD compared to VFAT, by running Postmark on a regular VFAT file system and on VFATTSD. Figure 6 shows the Postmark results for VFATTSD. The increase in wait time in VFATTSD (31%) is due to the increased seek times while updating FAT entries. This is because VFATTSD's FAT blocks are not contiguous. This is evident from the increased value of wuse for similar values of wsect.

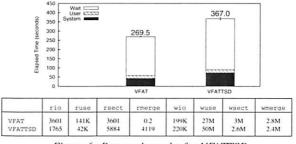


Figure 6: Postmark results for VFATTSD

8.3 ACCESS

We now discuss the results for Ext2ACCESS under Postmark, kernel compilation, and micro-benchmarks.

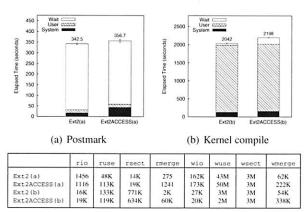


Figure 7: Postmark and Kernel compile results for Ext2ACCESS

Postmark. Figure 7(a) shows the Postmark results for Ext2ACCESS and Ext2. The overheads of Ext2-ACCESS are similar to Ext2TSD except that the system time overheads have increased significantly. Ext2ACCESS has 60% more system time than Ext2. This is because of the additional information such as keys and the temporal locality of reference blocks that ACCESS needs to track. Even though ACCESS incurs the overhead to write the capability table (Section 6.5) for each isegment, it has not affected the wait time because of better spatial locality.

Kernel compile. Figure 7(b) shows the results for the Ext2ACCESS kernel compile benchmark. Ext2ACCESS had 21% more system time than Ext2, due to the additional information being tracked by ACCESS. Ext2ACCESS's wait time was 2.5 times larger because of the increase in sleep time of the compile process context due to the preemption of the TSD commit thread. The sleep time has increased compared to Ext2TSD because ACCESS is more CPU-intensive than Ext2TSD.

Micro-benchmarks. Figures 3, 4, and 5 also show the results for the CREATE, LOOKUP, and the UNLINK micro-benchmarks for Ext2ACCESS, respectively. For the CREATE workload the system time and wait time of Ext2ACCESS increased by 44% and 2.2 times, respectively, compared to regular Ext2. The increase in wait time is because of two reasons. First, the I/O time for Ext2ACCESS is greater because of the additional seeks required to access the capability table. This is shown in the disk statistics. The wuse and ruse values of Ext2ACCESS are significantly higher than regular Ext2. Second, the preemption of the benchmark process context by the TSD commit thread has resulted in increased

sleep time. The results for the lookup workload are similar to that of Ext2TSD, except for the 12% increase in system time. For the unlink workload, Ext2ACCESS shows significant overheads: 90% more system time and 85% more wait time. This is because unlinking files involve several calls to the DELETE_PTR disk primitive which requires multiple hash table lookups. The increase in wait time is due to the increase in the time taken for writes as evidenced by the high wase value of disk statistics. This is because of the additional seeks required to update the capability tables for each isegment.

8.4 Secure Deletion

To evaluate the performance of our next case study (SDTSD), we ran an unlink micro-benchmark. Figure 8 shows the results of this benchmark. The I/O overhead of SDTSD over Ext2TSD was 40% compared to regular Ext2, mainly because of the additional I/O caused by overwrites for secure deletion. This is evidenced by the high wsect and wuse values for SDTSD, as expected.

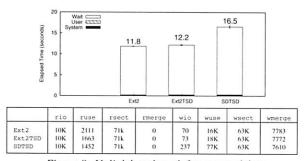


Figure 8: Unlink benchmark for secure delete

9 Related Work

Type-safety. The concept of type safety has been widely used in the context of programming languages. Type-safe languages such as Java are known to make programming easier by providing automatic memory management. More importantly, they improve security by restricting memory access to legal data structures. Type-safe languages use a philosophy very similar to our model: a capability to an encompassing data structure implies a capability to all entities enclosed within it. Type-safety has also been explored in the context of building secure operating systems. For example, the SPIN operating system [4] enabled safe kernel-level extensions by constraining them to be written in Modula-3, a type-safe language. Since the extension can only access objects it has explicit access to, it cannot change arbitrary kernel state. More recently, the Singularity operating system [15] used a similar approach, attempting to improve OS robustness and reliability by using typesafe languages and clearly defined interfaces.

Interface between file systems and disks. Our work is closely related to a large body of work examining new interfaces between file systems and storage. For example, logical disks expand the block-based interface by exposing a list-based mechanism that file systems use to convey grouping between blocks [7]. The Universal File Server [5] has two layers where the lower layer exists in the storage level, thereby conveying directory-file relationships to the storage layer. More recent research has suggested the evolution of the storage interface from the current block-based form to a higher-level abstraction. Object-based Storage Device (OSD) is one example [19]; in OSDs the disk manages variable-sized objects instead of blocks. Similar to TSD, object-based disks handle block allocation within an object, but still do not have information on the relationships across objects. Another example is Boxwood [18]; Boxwood considers making distributed file systems easier to develop by providing a distributed storage layer that exports higher-level data structures such as B-Trees. Unlike many of these interfaces, TSD considers backwards compatibility and ease of file system modification as an important goal. By following the block-based interface and augmenting it with minimal hooks, we enable file systems to be more readily portable to this interface, as this paper demonstrates. Others examine the storage interface by trying to keep the interface constant, but move some intelligence into the disk system. For example, the Loge disk controller implemented eager-writing by writing to a block closest to its disk arm [9]. The log-based programmable disk [29] extended this work, adding free-space compaction. These systems, while being easily deployable by not requiring interface change, are quite limited in the functionality they extend to disks.

A more recent example of work on improving storage functionality without changing the interface is Semantically-smart Disk Systems (SDSs) [27]. An SDS enables rich functionality by automatically tracking information about the file system or DBMS using the storage system, by carefully watching updates. However, semantic disks need to be tailored to the specifics of the file system above. In addition, they involve a fair amount of complexity to infer semantic information underneath asynchronous file systems. As the authors point out [25], SDS is valuable when the interface cannot be changed, but serves better as an evolutionary step towards an eventual change to an explicit interface such as TSD.

Capability-based access control. Network-Attached Secure Disks (NASDs) incorporate capability based access control in the context of distributed authentication using object-based storage [1,11,20]. Temporal timeouts in ACCESS are related to caching capabilities during a time interval in OSDs [2]. The notion of using a

single capability to access a group of blocks has been explored in previous research [1, 12, 21].

In contrast to their object-level capability enforcement, ACCESS uses implicit path-based capabilities using pointer relationships between blocks.

10 Conclusions

In this paper, we have taken the well-known concept of type-safety and applied it in the context of disk storage. We have explored a simple question: what can a disk do if it knew about pointers? We find that pointer information enables rich functionality within storage, and also enables better security through active enforcement of constraints within the disk system. We believe that this pointer abstraction explores an interesting and effective design choice in the large spectrum of work on alternative interfaces to storage.

Our experience with TSDs and the case studies has also pointed to some limitations with this approach. First, TSDs assume that a block is an atomic unit of file system structure. This assumption makes it hard to enforce constraints on data objects that occupy a partial block (e.g., multiple inodes per block). Second, the lack of higher level control over block allocation may limit the benefits of TSDs with software that need to place data in the exact physical locations on disk. While the current interface presents a reasonable choice, only future research will identify if more fine tuning is required.

11 Acknowledgments

We like to thank the anonymous reviewers for their helpful comments, and especially our shepherd Garth Gibson, whose meticulous and detailed comments helped improve the work. We thank Muthian Sivathanu for his valuable feedback during the various stages of this project. We would also like to thank the following people for their comments and suggestions on the work: Remzi H. Arpaci-Dusseau, Tzi-cker Chiueh, Christos Karamanolis, Patrick McDaniel, Ethan Miller, Abhishek Rai, R. Sekar, Radu Sion, Charles P. Wright, and the members of our research group (File systems and Storage Lab at Stony Brook).

This work was partially made possible by NSF CA-REER EIA-0133589 and NSF CCR-0310493 awards.

References

- [1] M. K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. A. Thekkath. Block-Level Security for Network-Attached Disks. In *Proc. of the Second USENIX Conf. on File and Storage Technologies*, pp. 159–174, San Francisco, CA, March 2003.
- [2] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and

- L. Yerushalmi. Towards an object store. In Mass Storage Systems and Technologies (MSST), 2003.
- [3] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In Proc. of the 10th Usenix Security Symposium, pp. 153-164, Washington, DC, August 2001.
- [4] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In Proc. of the 15th ACM Symposium on Operating System Principles, pp. 267-284, Copper Mountain Resort, CO, December 1995.
- [5] A. D. Birrell and R. M. Needham. A universal file server. In IEEE Transactions on Software Engineering, volume SE-6, pp. 450-453, September 1980.
- [6] M. Blaze. A Cryptographic File System for Unix. In Proc. of the first ACM Conf. on Computer and Communications Security, pp. 9-16, Fairfax, VA, 1993
- [7] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In Proc. of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, NY, October 2003.
- [8] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In Proc. of the Annual USENIX Technical Conf., pp. 177-190, Monterey, CA, June 2002.
- [9] R. English and A. Stepanov. Loge: A self-organizing disk controller. HP Labs, Tech. Rep., HPL91(179), 1991.
- [10] G. R. Ganger. Blurring the Line Between OSes and Storage Devices. Tech. Rep. CMU-CS-01-166, CMU, December 2001.
- [11] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In Proc. of the Eighth International Conf. on Architectural Support for Programming Langauges and Operating Systems (ASPLOS-VIII), pp. 92-103, New York, NY, December 1998
- [12] H. Gobioff. Security for a High Performance Commodity Storage Subsystem. PhD thesis, Carnegie Mellon University, May 1999. citeseer.ist.psu.edu/article/gobioff99security.html.
- [13] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In Proc. of the Sixth USENIX UNIX Security Symposium, pp. 77-90, San Jose, CA, July 1996.
- [14] V. Henson. Chunkfs and continuation inodes. The 2006 Linux Filesystems Workshop (Part III), 2006.
- [15] G. Hunt, J. Laurus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Tech. Rep. MSR-TR-2005-135, Microsoft Research, 2005.
- [16] N. Joukov and E. Zadok. Adding Secure Deletion to Your Favorite File System. In Proc. of the third international IEEE Security In Storage Workshop, San Fransisco, CA, December 2005.

- [17] J. Katcher. PostMark: A New Filesystem Benchmark. Tech. Rep. TR3022, Network Appliance, 1997. www. netapp.com/tech_library/3022.html.
- [18] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In Proc. of the 6th Symposium on Operating Systems Design and Implementation, pp. 105-120, San Francisco, CA, December 2004.
- [19] M. Mesnier, G. R. Ganger, and E. Riedel. Object based storage. IEEE Communications Magazine, 41, August 2003. ieeexplore.ieee.org.
- [20] E. Miller, W. Freeman, D. Long, and B. Reed. Strong Security for Network-Attached Storage. In Proc. of the First USENIX Conf. on File and Storage Technologies, pp. 1-13, Monterey, CA, January 2002.
- [21] E. L. Miller, W. E. Freeman, D. D. E. Long, and B. C. Reed. Strong security for network-attached storage. In USENIX Conf. on File and Storage Technologies (FAST), pp. 1-14, jan 2002.
- [22] Overwrite, Secure Deletion Software. www.kyuzz.org/ antirez/overwrite.
- [23] R. Perlman. Secure Deletion of Data. In Proc. of the third international IEEE Security In Storage Workshop, San Fransisco, CA, December 2005.
- [24] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S Jha. A Logic of File Systems. In Proc. of the Fourth USENIX Conf. on File and Storage Technologies, pp. 1-16, San Francisco, CA, December 2005.
- [25] M. Siyathanu, L. N. Bairayasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block-Level. In Proc. of the 6th Symposium on Operating Systems Design and Implementation, pp. 379-394, San Francisco, CA, December 2004.
- [26] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In Proc. of the Third USENIX Conf. on File and Storage Technologies, pp. 15-30, San Francisco, CA, March/April 2004.
- [27] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In Proc. of the Second USENIX Conf. on File and Storage Technologies, pp. 73-88, San Francisco, CA, March 2003.
- [28] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In Proc. of the 4th Usenix Symposium on Operating System Design and Implementation, pp. 165-180, San Diego, CA, October
- [29] R. Y. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log Based File Systems for a Programmable Disk. In Proc. of the Third Symposium on Operating Systems Design and Implementation, pp. 29-44, New Orleans, LA, February 1999.
- [30] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In Proc. of the Annual USENIX Technical Conf., pp. 197-210, San Antonio, TX, June 2003.

28

Stasis: Flexible Transactional Storage

Russell Sears and Eric Brewer

University of California, Berkeley {sears, brewer}@cs.berkeley.edu

An increasing range of applications requires robust support for atomic, durable and concurrent transactions. Databases provide the default solution, but force applications to interact via SQL and to forfeit control over data layout and access mechanisms. We argue there is a gap between DBMSs and file systems that limits designers of data-oriented applications.

Stasis is a storage framework that incorporates ideas from traditional write-ahead logging algorithms and file systems. It provides applications with flexible control over data structures, data layout, robustness, and performance. Stasis enables the development of unforeseen variants on transactional storage by generalizing write-ahead logging algorithms. Our partial implementation of these ideas already provides specialized (and cleaner) semantics to applications.

We evaluate the performance of a traditional transactional storage system based on Stasis, and show that it performs favorably relative to existing systems. We present examples that make use of custom access methods, modified buffer manager semantics, direct log file manipulation, and LSN-free pages. These examples facilitate sophisticated performance optimizations such as zero-copy I/O. These extensions are composable, easy to implement and significantly improve performance.

1 Introduction

As our reliance on computing infrastructure increases, a wider range of applications requires robust data management. Traditionally, data management has been the province of database management systems (DBMSs), which are well-suited to enterprise applications, but lead to poor support for systems such as web services, search engines, version systems, work-flow applications, bioinformatics, and scientific computing. These applications have complex transactional storage requirements, but do not fit well onto SQL or the monolithic approach of current databases. In fact, when performance matters these

applications often avoid DBMSs and instead implement ad-hoc data management solutions [15, 17].

An example of this mismatch occurs with DBMS support for persistent objects. In a typical usage, an array of objects is made persistent by mapping each object to a row in a table (or sometimes multiple tables) [22] and then issuing queries to keep the objects and rows consistent. Also, for efficiency, most systems must buffer two copies of the application's working set in memory. This is an awkward and inefficient mechanism, and hence we claim that DBMSs do not support this task well.

Search engines and data warehouses in theory can use the relational model, but in practice need a very different implementation. Object-oriented, XML, and streaming databases all have distinct conceptual models and underlying implementations.

Scientific computing, bioinformatics and document management systems tend to preserve old versions and track provenance. Thus they each have a distinct conceptual model. Bioinformatics systems perform computations over large, semi-structured databases. Relational databases support none of these requirements well. Instead, office suites, ad-hoc text-based formats and Perl scripts are used for data management [48], with mixed success [57].

Our hypothesis is that 1) each of these areas has a distinct top-down conceptual model (which may not map well to the relational model); and 2) there exists a bottom-up layered framework that can better support all of these models and others.

To explore this hypothesis, we present Stasis, a library that provides transactional storage at a level of abstraction as close to the hardware as possible. It can support special-purpose transactional storage models in addition to ACID database-style interfaces to abstract data models. Stasis incorporates techniques from both databases (e.g. write-ahead logging) and operating systems (e.g. zero-copy techniques).

Our goal is to combine the flexibility and layering of

low-level abstractions typical for systems work with the complete semantics that exemplify the database field. By *flexible* we mean that Stasis can support a wide range of transactional data structures *efficiently*, and that it can support a variety of policies for locking, commit, clusters and buffer management. Also, it is extensible for new core operations and data structures. This flexibility allows it to support a wide range of systems and models.

By complete we mean full redo/undo logging that supports both no force, which provides durability with only log writes, and steal, which allows dirty pages to be written out prematurely to reduce memory pressure. By complete, we also mean support for media recovery, which is the ability to roll forward from an archived copy, and support for error-handling, clusters, and multithreading. These requirements are difficult to meet and form the raison d'être for Stasis: the framework delivers these properties as reusable building blocks for systems that implement complete transactions.

Through examples and their good performance, we show how Stasis efficiently supports a wide range of uses that fall in the gap between database and file system technologies, including persistent objects, graph- or XML-based applications, and recoverable virtual memory [42].

For example, on an object persistence workload, we provide up to a 4x speedup over an in-process MySQL implementation and a 3x speedup over Berkeley DB, while cutting memory usage in half (Section 5.3). We implemented this extension in 150 lines of C, including comments and boilerplate. We did not have this type of optimization in mind when we wrote Stasis, and in fact the idea came from a user unfamiliar with Stasis.

This paper begins by contrasting Stasis' approach with that of conventional database and transactional storage systems. It proceeds to discuss write-ahead logging, and describe ways in which Stasis can be customized to implement many existing (and some new) write-ahead logging variants. We present implementations of some of these variants and benchmark them against popular realworld systems. We conclude with a survey of related and future work.

An (early) open-source implementation of the ideas presented here is available (see Section 10).

2 Stasis is not a Database

Database research has a long history, including the development of many of the technologies we exploit. This section explains why databases are fundamentally inappropriate tools for system developers, and covers some of the previous responses of the systems community. These problems have been the focus of database and systems researchers for at least 25 years.

2.1 The Database View

The database community approaches the limited range of DBMSs by either creating new top-down models, such as object-oriented, XML or streaming databases [11, 28, 33], or by extending the relational model [14] along some axis, such as new data types [50]. We cover these attempts in more detail in Section 6.

An early survey of database implementations sought to enumerate the components used by database system implementors [4, 6]. This survey was performed due to difficulties in extending database systems into new application domains. It divided internal database routines into two broad modules: *conceptual mappings* and *physical database models*. It is the responsibility of a database implementor to choose a set of conceptual mappings that implement the desired higher-level abstraction (such as the relational model). The physical data model is chosen to support efficiently the set of mappings that are built on top of it.

A conceptual mapping based on the relational model might translate a relation into a set of keyed tuples. If the database were going to be used for short, write-intensive and high-concurrency transactions (e.g. banking), the physical model would probably translate sets of tuples into an on-disk B-tree. In contrast, if the database needed to support long-running, read-only aggregation queries over high-dimensional data (e.g. data warehousing), a physical model that stores the data in a sparse array format would be more appropriate [12, 58]. Although both kinds of databases are based upon the relational model they make use of different physical models in order to serve different classes of applications efficiently.

A basic claim of this paper is that no known physical data model can efficiently support the wide range of conceptual mappings that are in use today. In addition to sets, objects, and XML, such a model would need to cover search engines, version-control systems, workflow applications, and scientific computing, as examples. Similarly, a recent database paper argues that the "one size fits all" approach of DBMSs no longer works [51].

Instead of attempting to create such a unified model after decades of database research has failed to produce one, we opt to provide a bottom-up transactional toolbox that supports many models efficiently. This makes it easy for system designers to implement most data models that the underlying hardware can support, or to abandon the database approach entirely, and forgo a top-down model.

2.2 The Systems View

The systems community has also worked on this mismatch, which has led to many interesting projects. Examples include alternative durability models such as QuickSilver [43], RVM [42], persistent objects [29], and persistent data structures [20, 32]. We expect that Stasis would simplify the implementation of most if not all of these systems. Section 6 covers these in more detail.

In some sense, our hypothesis is trivially true in that there exists a bottom-up framework called the "operating system" that can implement all of the models. A famous database paper argues that it does so poorly [49]. Our task is really to simplify the implementation of transactional systems through more powerful primitives that enable concurrent transactions with a variety of performance/robustness tradeoffs.

The closest system to ours in spirit is Berkeley DB, a highly successful lightweight alternative to conventional databases [45]. At its core, it provides the physical database model (relational storage system [3]) of a conventional database server. In particular, it provides transactional (ACID) operations on B-trees, hash tables, and other access methods. It provides flags that let its users tweak aspects of the performance of these primitives, and selectively disable the features it provides.

With the exception of the benchmark designed to compare the two systems, none of the Stasis applications presented in Section 5 are efficiently supported by Berkeley DB. This is a result of Berkeley DB's assumptions regarding workloads and low-level data representations. Thus, although Berkeley DB could be built on top of Stasis, Berkeley DB's data model and write-ahead logging system are too specialized to support Stasis.

3 Transactional Pages

This section describes how Stasis implements transactions that are similar to those provided by relational database systems, which are based on transactional pages. The algorithms described in this section are not novel, and are in fact based on ARIES [35]. However, they form the starting point for extensions and novel variants, which we cover in the next two sections.

As with other systems, Stasis' transactions have a multi-level structure. Multi-layered transactions were originally proposed as a concurrency control strategy for database servers that support high-level, application-specific extensions [55]. In Stasis, the lower level of an operation provides atomic updates to regions of the disk. These updates do not have to deal with concurrency, but must update the page file atomically, even if the system crashes.

Higher-level operations span multiple pages by atomically applying sets of operations to the page file, recording their actions in the log and coping with concurrency issues. The loose coupling of these layers lets Stasis' users compose and reuse existing operations.

3.1 Atomic Disk Operations

Transactional storage algorithms work by atomically updating portions of durable storage. These small atomic updates bootstrap transactions that are too large to be applied atomically. In particular, write-ahead logging (and therefore Stasis) relies on the ability to write entries to the log file atomically. Transaction systems that store sequence numbers on pages to track versions rely on atomic page writes in addition to atomic log writes.

In practice, a write to a disk page is not atomic (in modern drives). Two common failure modes exist. The first occurs when the disk writes a partial sector during a crash. In this case, the drive maintains an internal checksum, detects a mismatch, and reports it when the page is read. The second case occurs because pages span multiple sectors. Drives may reorder writes on sector boundaries, causing an arbitrary subset of a page's sectors to be updated during a crash. *Torn page detection* can be used to detect this phenomenon, typically by requiring a checksum for the whole page.

Torn and corrupted pages may be recovered by using *media recovery* to restore the page from backup. Media recovery works by reloading the page from an archive copy, and bringing it up to date by replaying the log.

For simplicity, this section ignores mechanisms that detect and restore torn pages, and assumes that page writes are atomic. We relax this restriction in Section 4.

3.2 Non-concurrent Transactions

This section provides the "Atomicity" and "Durability" properties for a single ACID transaction. First we describe single-page transactions, then multi-page transactions. "Consistency" and "Isolation" are covered with concurrent transactions in the next section.

The insight behind transactional pages was that atomic page writes form a good foundation for full transactions. However, since page writes are no longer atomic, it might be better to think of these as transactional sectors.

The trivial way to achieve single-page transactions is to apply all of the updates to the page and then write it out on commit. The page must be pinned until commit to prevent write-back of uncommitted data, but no logging is required.

This approach performs poorly because we *force* the page to disk on commit, which leads to a large number of synchronous non-sequential writes. By writing redo information to the log before committing (write-ahead logging), we get *no-force* transactions and better performance, since the synchronous writes to the log are sequential. Later, the pages are written out asynchronously, often as part of a larger sequential write.

After a crash, we have to apply the redo entries to

those pages that were not updated on disk. To decide which updates to reapply, we use a per-page version number called the *log-sequence number* or *LSN*. Each update to a page increments the LSN, writes it on the page, and includes it in the log entry. On recovery, we load the page, use the LSN to figure out which updates are missing (those with higher LSNs), and reapply them.

Updates from aborted transactions should not be applied, so we also need to log commit records; a transaction commits when its commit record correctly reaches the disk. Recovery starts with an analysis phase that determines all of the outstanding transactions and their fate. The redo phase then applies the missing updates for committed transactions.

Pinning pages until commit also hurts performance, and could even affect correctness if a single transaction needs to update more pages than can fit in memory. A related problem is that with concurrency a single page may be pinned forever as long as it has at least one active transaction in progress all the time. Systems that support *steal* avoid these problems by allowing pages to be written back early. This implies we may need to undo updates on the page if the transaction aborts, and thus before we can write out the page we must write the undo information to the log.

On recovery, the redo phase applies all updates (even those from aborted transactions). Then, an undo phase corrects stolen pages for aborted transactions. Each operation that undo performs is recorded in the log, and the per-page LSN is updated accordingly. In order to ensure progress even with crashes during recovery, special log records mark which actions have been undone, so they may be skipped during recovery in the future. We also use these records, called *Compensation Log Records* (*CLRs*) to avoid undoing actions that we intend to keep even when transactions abort.

The primary difference between Stasis and ARIES for basic transactions is that Stasis allows user-defined operations, while ARIES defines a set of operations that support relational database systems. An *operation* consists of an undo and a redo function. Each time an operation is invoked, a corresponding log entry is generated. We describe operations in more detail in Section 3.4.

Given steal/no-force single-page transactions, it is relatively easy to build full transactions. To recover a multipage transaction, we simply recover each of the pages individually. This works because steal/no-force completely decouples the pages: any page can be written back early (steal) or late (no-force).

3.3 Concurrent Transactions

Two factors make it more complicated to write operations that may be used in concurrent transactions. The first is familiar to anyone that has written multi-threaded code: Accesses to shared data structures must be protected by latches (mutexes). The second problem stems from the fact that abort cannot simply roll back physical updates. Fortunately, it is straightforward to reduce this second, transaction-specific problem to the familiar problem of writing multi-threaded software.

To understand the problems that arise with concurrent transactions, consider what would happen if one transaction, A, rearranges the layout of a data structure. Next, another transaction, B, modifies that structure and then A aborts. When A rolls back, its undo entries will undo the changes that it made to the data structure, without regard to B's modifications. This is likely to cause corruption.

Two common solutions to this problem are total isolation and nested top actions. Total isolation prevents any transaction from accessing a data structure that has been modified by another in-progress transaction. An application can achieve this using its own concurrency control mechanisms, or by holding a lock on each data structure until the end of the transaction (by performing strict two-phase locking on the entire data structure). Releasing the lock after the modification, but before the end of the transaction, increases concurrency. However, it means that follow-on transactions that use the data may need to abort if this transaction aborts (cascading aborts).

Nested top actions avoid this problem. The key idea is to distinguish between the logical operations of a data structure, such as adding an item to a set, and internal physical operations such as splitting tree nodes. The internal operations do not need to be undone if the containing transaction aborts; instead of removing the data item from the page, and merging any nodes that the insertion split, we simply remove the item from the set as application code would—we call the data structure's *remove* method. That way, we can undo the insertion even if the nodes that were split no longer exist, or if the data item has been relocated to a different page. This lets other transactions manipulate the data structure before the first transaction commits.

In Stasis, each nested top action performs a single logical operation by applying a number of physical operations to the page file. Physical redo and undo log entries are stored in the log so that recovery can repair any temporary inconsistency that the nested top action introduces. Once the nested top action has completed, a logical undo entry is recorded, and a CLR is used to tell recovery and abort to skip the physical undo entries.

This leads to a mechanical approach for creating reentrant, concurrent operations:

1. Wrap a mutex around each operation. With care, it is possible to use finer-grained latches in a Stasis operation [36], but it is rarely necessary.

- Define a *logical* undo for each operation (rather than a set of page-level undos). For example, this is easy for a hash table: the undo for *insert* is *remove*. The logical undo function should arrange to acquire the mutex when invoked by abort or recovery.
- Add a "begin nested top action" right after mutex acquisition, and an "end nested top action" right before mutex release. Stasis includes operations that provide nested top actions.

If the transaction that encloses a nested top action aborts, the logical undo will *compensate* for the effects of the operation, taking updates from concurrent transactions into account. Using this recipe, it is relatively easy to implement thread-safe concurrent transactions. Therefore, they are used throughout Stasis' default data structure implementations. This approach also works with the variable-sized atomic updates covered in Section 4.

3.4 User-Defined Operations

The first kind of extensibility enabled by Stasis is userdefined operations. Figure 1 shows how operations interact with Stasis. A number of default operations come with Stasis. These include operations that allocate and manipulate records, operations that implement hash tables, and a number of methods that add functionality to recovery. Many of the customizations described below are implemented using custom operations.

In this portion of the discussion, physical operations are limited to a single page, as they must be applied atomically. Section 4 removes this constraint.

Operations are invoked by registering a callback (the "operation implementation" in Figure 1) with Stasis at startup, and then calling Tupdate() to invoke the operation at runtime. Stasis ensures that operations follow the write-ahead logging rules required for steal/no-force transactions by controlling the timing and ordering of log and page writes.

The redo log entry consists of the LSN and an argument that will be passed to redo. The undo entry is analogous.² Each operation should be deterministic, provide an inverse, and acquire all of its arguments from the argument passed via Tupdate(), from the page it updates, or both. The callbacks used during forward operation are also used during recovery. Therefore operations provide a single redo function and a single undo function. There is no "do" function, which reduces the amount of recovery-specific code in the system.

The first step in implementing a new operation is to decide upon an external interface, which is typically cleaner than directly calling Tupdate() to invoke the operation(s). The externally visible interface is implemented by wrapper functions and read-only access meth-

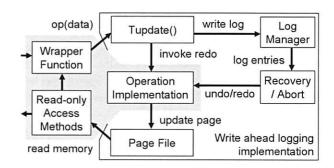


Figure 1: The portions of Stasis that directly interact with new operations. Arrows point in the direction of data flow.

ods. The wrapper function modifies the state of the page file by packaging the information that will be needed for redo/undo into a data format of its choosing. This data structure is passed into Tupdate(), which writes a log entry and invokes the redo function.

The redo function modifies the page file directly (or takes some other action). It is essentially an interpreter for its log entries. Undo works analogously, but is invoked when an operation must be undone.

This pattern applies in many cases. In order to implement a "typical" operation, the operation's implementation must obey a few more invariants:

- Pages should only be updated inside physical redo and undo operation implementations.
- Logical operations may invoke other operations via Tupdate(). Recovery does not support logical redo, and physical operation implementations may not invoke Tupdate().
- The page's LSN should be updated to reflect the changes (this is generally handled by passing the LSN to the page implementation).
- Nested top actions (and logical undo) or "big locks" (total isolation) should be used to manage concurrency (Section 3.3).

Although these restrictions are not trivial, they are not a problem in practice. Most read-modify-write actions can be implemented as user-defined operations, including common DBMS optimizations such as increment operations, and many optimizations based on ARIES [26, 36]. The power of Stasis is that by following these local restrictions, operations meet the global invariants required by correct, concurrent transactions.

Finally, for some applications, the overhead of logging information for redo or undo may outweigh their benefits. Operations that wish to avoid undo logging can call an API that pins the page until commit, and use an empty undo function. Similarly, forcing a page to be written out on commit avoids redo logging.

3.5 Application-specific Locking

The transactions described above provide the "Atomicity" and "Durability" properties of ACID. "Isolation" is typically provided by locking, which is a higher level but compatible layer. "Consistency" is less well defined but comes in part from low-level mutexes that avoid races, and in part from higher-level constructs such as unique key requirements. Stasis and most databases support this by distinguishing between *latches* and *locks*. Latches are provided using OS mutexes, and are held for short periods of time. Stasis' default data structures use latches in a way that does not deadlock. This allows higher-level code to treat Stasis as a conventional reentrant data structure library.

This section describes Stasis' latching protocols and describes two custom lock managers that Stasis' allocation routines use. Applications that want conventional transactional isolation (serializability) can make use of a lock manager or optimistic concurrency control [1, 27]. Alternatively, applications may follow the example of Stasis' default data structures, and implement deadlock prevention, or other custom lock management schemes.

Note that locking schemes may be layered as long as no legal sequence of calls to the lower level results in deadlock, or the higher level is prepared to handle deadlocks reported by the lower levels.

When Stasis allocates a record, it first calls a region allocator, which allocates contiguous sets of pages, and then it allocates a record on one of those pages. The record allocator and the region allocator each contain custom lock management. The lock management prevents one transaction from reusing storage freed by another, active transaction. If this storage were reused and then the transaction that freed it aborted, then the storage would be double-allocated.

The region allocator, which allocates large chunks infrequently, records the id of the transaction that created a region of freespace, and does not coalesce or reuse any storage associated with an active transaction. In contrast, the record allocator is called frequently and must enable locality. It associates a set of pages with each transaction, and keeps track of deallocation events, making sure that space on a page is never overbooked. Providing each transaction with a separate pool of freespace increases concurrency and locality. This is similar to Hoard [7] and McRT-malloc [23] (Section 6.4).

Note that both lock managers have implementations that are tied to the code they service, both implement deadlock avoidance, and both are transparent to higher layers. General-purpose database lock managers provide none of these features, supporting the idea that special-purpose lock managers are a useful abstraction. Locking schemes that interact well with object-oriented program-

ming schemes [44] and exception handling [24] extend these ideas to larger systems.

Although custom locking is important for flexibility, it is largely orthogonal to the concepts described in this paper. We make no assumptions regarding lock managers being used by higher-level code in the remainder of this discussion.

4 LSN-free Pages

The recovery algorithm described above uses LSNs to determine the version number of each page during recovery. This is a common technique. As far as we know, it is used by all database systems that update data in place. Unfortunately, this makes it difficult to map large objects onto pages, as the LSNs break up the object. It is tempting to store the LSNs elsewhere, but then they would not be updated atomically, which defeats their purpose.

This section explains how we can avoid storing LSNs on pages in Stasis without giving up durable transactional updates. The techniques here are similar to those used by RVM [42], a system that supports transactional updates to virtual memory. However, Stasis generalizes the concept, allowing it to coexist with traditional pages and more easily support concurrent transactions.

In the process of removing LSNs from pages, we are able to relax the atomicity assumptions that we make regarding writes to disk. These relaxed assumptions allow recovery to repair torn pages without performing media recovery, and allow arbitrary ranges of the page file to be updated by a single physical operation.

Stasis' implementation does not currently support the recovery algorithm described in this section. However, Stasis avoids hard-coding most of the relevant subsystems. LSN-free pages are essentially an alternative protocol for atomically and durably applying updates to the page file. This will require the addition of a new page type that calls the logger to estimate LSNs; Stasis currently has three such types, and already supports the coexistence of multiple page types within the same page file or logical operation.

4.1 Blind Updates

Recall that LSNs were introduced to allow recovery to guarantee that each update is applied exactly once. This was necessary because some operations that manipulate pages are not idempotent, or simply make use of state stored in the page.

As described above, Stasis operations may make use of page contents to compute the updated value, and Stasis ensures that each operation is applied exactly once in the right order. The recovery scheme described in this section does not guarantee that operations will be applied

Recovery's initial LSN estimates
Dirty page table from log, LSN=9

Page #							
0	1	2	3	4	5		
0	0	0	0	0	0		
Dir	ty P	age	R	RecLSN			
	3			3			
	0			8			
	2			5			
7	9	4	2	9	9		

Updated LSN estimates

Figure 2: LSN estimation. If a page was not mentioned in the log, it must have been up-to-date on disk. RecLSN is the LSN of the entry that caused the page to become dirty. Subtracting one gives us a safe estimate of the page

exactly once, or even that they will be presented with a self-consistent version of a page during recovery.

Therefore, in this section we focus on operations that produce deterministic, idempotent redo entries that do not examine page state. We call such operations *blind updates*. For example, a blind update's operation could use log entries that contain a set of byte ranges with their new values. Note that we still allow code that invokes operations to examine the page file, just not during the redo phase of recovery.

Recovery works the same way as before, except that it now estimates page LSNs rather than reading them from pages. One safe estimate is the LSN of the most recent archive or log truncation point. Alternatively, Stasis could occasionally store its *dirty page table* to the log (Figure 2). The dirty page table lists all dirty pages and their *recovery LSNs*. It is used by ARIES to reduce the amount of work that must be performed during REDO.

The recovery LSN (RecLSN) is the LSN of the log entry that caused a clean (up-to-date on disk) page to become dirty. No log entries older than the RecLSN need to be applied to the page during redo. Therefore, redo can safely estimate the page LSN by choosing any number less than RecLSN. If a page is not in the table, redo can use the LSN of the log entry that contains the table, since the page must have been clean when the log entry was produced. Stasis writes the dirty page table to log whether or not LSN-free pages are in use, so we expect the runtime overhead to be negligible.

Although the mechanism used for recovery is similar, the invariants maintained during recovery have changed. With conventional transactions, if a page in the page file is internally consistent immediately after a crash, then the page will remain internally consistent throughout the recovery process. This is not the case with our LSN-free scheme. Internal page inconsistencies may be introduced because recovery has no way of knowing the exact version of a page. Therefore, it may overwrite new portions of a page with older data from the log. The page

will then contain a mixture of new and old bytes, and any data structures stored on the page may be inconsistent. However, once the redo phase is complete, any old bytes will be overwritten by their most recent values, so the page will return to a self-consistent up-to-date state. (Section 4.4 explains this in more detail.)

Undo is unaffected except that any redo records it produces must be blind updates just like normal operation. We don't expect this to be a practical problem.

The rest of this section describes how concurrent, LSN-free pages allow standard file system and database optimizations to be easily combined, and shows that the removal of LSNs from pages simplifies recovery while increasing its flexibility.

4.2 Zero-copy I/O

We originally developed LSN-free pages as an efficient method for transactionally storing and updating multipage objects, called *blobs*. If a large object is stored in pages that contain LSNs, then it is not contiguous on disk, and must be gathered together by using the CPU to do an expensive copy into a second buffer.

In contrast, modern file systems allow applications to perform a DMA copy of the data into memory, allowing the CPU to be used for more productive purposes. Furthermore, modern operating systems allow network services to use DMA and network-interface cards to read data from disk, and send it over the network without passing it through the CPU. Again, this frees the CPU, allowing it to perform other tasks.

We believe that LSN-free pages will allow reads to make use of such optimizations in a straightforward fashion. Zero-copy writes are more challenging, but the goal would be to use one sequential write to put the new version on disk and then update metadata accordingly. We need not put the blob in the log if we avoid update in place; most blob implementations already avoid update in place since the length may vary between writes. We suspect that contributions from log-based file systems [41] can address these issues. In particular, we imagine writing large blobs to a distinct log segment and just entering metadata in the primary log.

4.3 Concurrent RVM

LSN-free pages are similar to the recovery scheme used by recoverable virtual memory (RVM) and Camelot [16]. RVM used purely physical logging and LSN-free pages so that it could use mmap to map portions of the page file into application memory [42]. However, without support for logical log entries and nested top actions, it is difficult to implement a concurrent, durable data structure using RVM or Camelot. (The description of Argus in Section 6.2.2 sketches one approach.)

In contrast, LSN-free pages allow logical undo and therefore nested top actions and concurrent transactions; a concurrent data structure need only provide Stasis with an appropriate inverse each time its logical state changes.

We plan to add RVM-style transactional memory to Stasis in a way that is compatible with fully concurrent in-memory data structures such as hash tables and trees, and with existing Stasis data structure implementations.

4.4 Unbounded Atomicity

Unlike transactions with per-page LSNs, transactions based on blind updates do not require atomic page writes and thus impose no meaningful boundaries on atomic updates. We still use pages to simplify integration into the rest of the system, but need not worry about torn pages. In fact, the redo phase of the LSN-free recovery algorithm effectively creates a torn page each time it applies an old log entry to a new page. However, it guarantees that all such torn pages will be repaired by the time redo completes. In the process, it also repairs any pages that were torn by a crash. This also implies that blind-update transactions work with storage technologies with different (and varying or unknown) units of atomicity.

Instead of relying upon atomic page updates, LSN-free recovery relies on a weaker property, which is that each bit in the page file must be either:

- 1. The version that was being overwritten at the crash.
- 2. The newest version of the bit written to storage.
- 3. Detectably corrupt (the storage hardware issues an error when the bit is read).

Modern drives provide these properties at a sector level: Each sector is updated atomically, or it fails a checksum when read, triggering an error. If a sector is found to be corrupt, then media recovery can be used to restore the sector from the most recent backup.

To ensure that we correctly update all of the old bits, we simply play the log forward from a point in time that is known to be older than the LSN of the page (which we must estimate). For bits that are overwritten, we end up with the correct version, since we apply the updates in order. For bits that are not overwritten, they must have been correct before and remain correct after recovery. Since all operations performed by redo are blind updates, they can be applied regardless of whether the initial page was the correct version or even logically consistent.

Figure 3 describes a page that is torn during crash, and the actions performed by redo that repair it. Assume that the initial version of the page, with LSN 0, is on disk, and the OS is in the process of writing out the version with

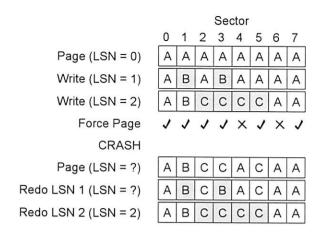


Figure 3: Torn pages and LSN-free recovery. The page is torn during the crash, but consistent once redo completes. Overwritten sectors are shaded.

LSN 2 when the system crashes. When recovery reads the page from disk, it may encounter any combination of sectors from these two versions.

Note that sectors zero, six and seven are not overwritten by any of the log entries that Redo will play back. Therefore, their values are unchanged in both versions of the page. In the example, zero and seven are overwritten during the crash, while six is left over from the old version of the page.

Redoing LSN 1 is unnecessary, since all of its sectors happened to make it to disk. However, recovery has no way of knowing this and applies the entry to the page, replacing sector three with an older version. When LSN 2 is applied, it brings this sector up to date, and also overwrites sector four, which did not make it to disk. At this point, the page is internally consistent.

Since LSN-free recovery only relies upon atomic updates at the bit level, it decouples page boundaries from atomicity and recovery. This allows operations to manipulate atomically (potentially non-contiguous) regions of arbitrary size by producing a single log entry. If this log entry includes a logical undo function (rather than a physical undo), then it can serve the purpose of a nested top action without incurring the extra log bandwidth of storing physical undo information. Such optimizations can be implemented using conventional transactions, but they appear to be easier to implement and reason about when applied to LSN-free pages.

4.5 Summary

In these last two sections, we explored some of the flexibility of Stasis. This includes user-defined operations, combinations of steal and force on a per-operation basis, flexible locking options, and a new class of transactions based on blind updates that enables better support for DMA, large objects, and multi-page operations. In

the next section, we show through experiments how this flexibility enables important optimizations and a widerange of transactional systems.

5 Experiments

Stasis provides applications with the ability to customize storage routines and recovery semantics. In this section, we show that this flexibility does not come with a significant performance cost for general-purpose transactional primitives, and show how a number of special-purpose interfaces aid in the development of higher-level code while significantly improving application performance.

5.1 Experimental setup

We chose Berkeley DB in the following experiments because it provides transactional storage primitives similar to Stasis, is commercially maintained and is designed for high performance and high concurrency. For all tests, the two libraries provide the same transactional semantics unless explicitly noted.

All benchmarks were run on an Intel Xeon 2.8 GHz processor with 1GB of RAM and a 10K RPM SCSI drive using ReiserFS [40].³ All results correspond to the mean of multiple runs with a 95% confidence interval with a half-width of 5%.

Our experiments use Berkeley DB 4.2.52 with the flags DB_TXN_SYNC (force log to disk on commit), and DB_THREAD (thread safety) enabled. We increased Berkeley DB's buffer cache and log buffer sizes to match Stasis' default sizes. If Berkeley DB implements a feature that Stasis is missing we enable it if it improves performance.

We disable Berkeley DB's lock manager for the benchmarks, though we use "Free Threaded" handles for all tests. This significantly increases performance by eliminating transaction deadlock, abort, and repetition. However, disabling the lock manager caused concurrent Berkeley DB benchmarks to become unstable, suggesting either a bug or misuse of the feature. With the lock manager enabled, Berkeley DB's performance in the multi-threaded benchmark (Section 5.2) strictly decreased with increased concurrency.

We expended a considerable effort tuning Berkeley DB and our efforts significantly improved Berkeley DB's performance on these tests. Although further tuning by Berkeley DB experts would probably improve Berkeley DB's numbers, we think our comparison shows that the systems' performance is comparable. As we add functionality, optimizations, and rewrite modules, Stasis' relative performance varies. We expect Stasis' extensions and custom recovery mechanisms to continue to perform similarly to comparable monolithic implementations.

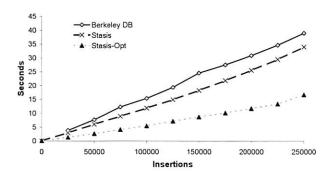


Figure 4: Performance of Stasis and Berkeley DB hash table implementations. The test is run as a single transaction, minimizing synchronous log writes.

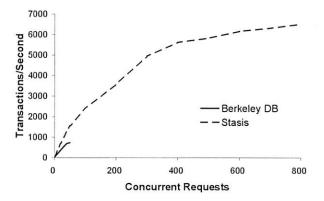


Figure 5: High-concurrency hash table performance. Our Berkeley DB test can only support 50 threads (see text).

5.2 Linear hash table

This section presents two hash table implementations built on top of Stasis, and compares them with the hash table provided by Berkeley DB. One of the Stasis implementations is simple and modular, while the other is monolithic and hand-tuned. Our experiments show that Stasis' performance is competitive, both with single-threaded and high-concurrency transactions.

The modular hash table uses nested top actions to update its internal structure atomically. It uses a *linear* hash function [30], allowing it to increase capacity incrementally. It is based on a number of modular subcomponents. Notably, the physical location of each bucket is stored in a growable array of fixed-length entries. This data structure is similar to Java's ArrayList. The bucket lists can be provided by either of Stasis' two linked list implementations. The first provides fixed-length entries, yielding a hash table with fixed-length keys and values. Our experiments use the second implementation, which provides variable-length entries (and therefore variable-length keys and values).

The hand-tuned hash table is also built on Stasis and also uses a linear hash function. However, it is monolithic and uses carefully ordered writes to reduce runtime overheads such as log bandwidth. Berkeley DB's hash table is a popular, commonly deployed implementation, and serves as a baseline for our experiments.

Both of our hash tables outperform Berkeley DB on a workload that populates the tables by repeatedly inserting (key, value) pairs (Figure 4). The performance of the modular hash table shows that data structure implementations composed from simpler structures can perform comparably to the implementations included in existing monolithic systems. The hand-tuned implementation shows that Stasis allows application developers to optimize important primitives.

Figure 5 describes the performance of the two systems under highly concurrent workloads using the ext3 file system.⁴ For this test, we used the modular hash table, since we are interested in the performance of a simple, clean data structure implementation that a typical system implementor might produce, not the performance of our own highly tuned implementation.

Both Berkeley DB and Stasis can service concurrent calls to commit with a single synchronous I/O. Stasis scaled quite well, delivering over 6000 transactions per second, and provided roughly double Berkeley DB's throughput (up to 50 threads). Although not shown here, we found that the latencies of Berkeley DB and Stasis were similar.

5.3 Object persistence

Two different styles of object persistence have been implemented on top of Stasis. The first object persistence mechanism, pobj, provides transactional updates to objects in Titanium, a Java variant. It transparently loads and persists entire graphs of objects, but will not be discussed in further detail. The second variant was built on top of a C++ object persistence library, Oasys. Oasys uses plug-in storage modules that implement persistent storage, and includes plugins for Berkeley DB and MySQL. Like C++ objects, Oasys objects are explicitly freed. However, Stasis could also support concurrent and incremental atomic garbage collection [26].

This section describes how the Stasis plugin supports optimizations that reduce the amount of data written to log and halve the amount of RAM required. We present three variants of the Stasis plugin. The basic one treats Stasis like Berkeley DB. The "update/flush" variant customizes the behavior of the buffer manager. Finally, the "delta" variant uses update/flush, but only logs the differences between versions.

The update/flush variant allows the buffer manager's view of live application objects to become stale. This is safe since the system is always able to reconstruct the appropriate page entry from the live copy of the object. This reduces the number of times the plugin must update

serialized objects in the buffer manager, and allows us to nearly eliminate the memory used by the buffer manager.

We implemented the Stasis buffer pool optimization by adding two new operations, update(), which updates the log when objects are modified, and flush(), which updates the page when an object is evicted from the application's cache.

The reason it would be difficult to do this with Berkeley DB is that we still need to generate log entries as the object is being updated. This would cause Berkeley DB to write data to pages, increasing the working set of the program and the amount of disk activity.

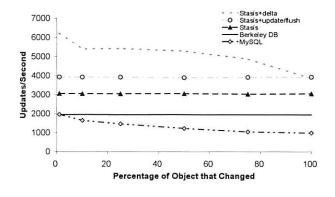
Furthermore, Stasis' copy of the objects is updated in the order objects are evicted from cache, not the update order. Therefore, the version of each object on a page cannot be determined from a single LSN.

We solve this problem by using blind updates to modify objects in place, but maintain a per-page LSN that is updated whenever an object is allocated or deallocated. At recovery, we apply allocations and deallocations based on the page LSN. To redo an update, we first decide whether the object that is being updated exists on the page. If so, we apply the blind update. If not, then the object must have been freed, so we do not apply the update. Because support for blind updates is only partially implemented, the experiments presented below mimic this behavior at runtime, but do not support recovery.

We also considered storing multiple LSNs per page and registering a callback with recovery to process the LSNs. However, in such a scheme, the object allocation routine would need to track objects that were deleted but still may be manipulated during redo. Otherwise, it could inadvertently overwrite per-object LSNs that would be needed during recovery. Alternatively, we could arrange for the object pool to update atomically the buffer manager's copy of all objects that share a given page.

The third plugin variant, "delta," incorporates the update/flush optimizations, but only writes changed portions of objects to the log. With Stasis' support for custom log formats, this optimization is straightforward.

Oasys does not provide a transactional interface. Instead, it is designed to be used in systems that stream objects over an unreliable network connection. The objects are independent of each other, so each update should be applied atomically. Therefore, there is never any reason to roll back an applied object update. Furthermore, Oasys provides a sync method, which guarantees the durability of updates after it returns. In order to match these semantics as closely as possible, Stasis' update/flush and delta optimizations do not write any undo information to the log. The Oasys sync method is implemented by committing the current Stasis transaction, and beginning a new one.



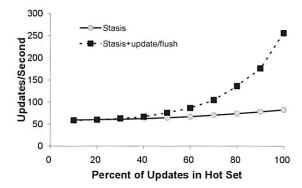


Figure 6: The effect of Stasis object-persistence optimizations under low and high memory pressure.

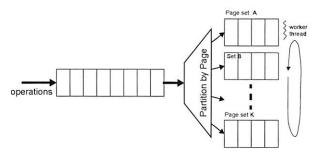


Figure 7: Locality-based request reordering. Requests are partitioned into queues. Queue are handled independently, improving locality and allowing requests to be merged.

As far as we can tell, MySQL and Berkeley DB do not support this optimization in a straightforward fashion. "Auto-commit" comes close, but does not quite provide the same durability semantics as Oasys' explicit syncs.

The operations required for the update/flush and delta optimizations required 150 lines of C code, including whitespace, comments and boilerplate function registrations.⁵ Although the reasoning required to ensure the correctness of this optimization is complex, the simplicity of the implementation is encouraging.

In this experiment, Berkeley DB was configured as described above. We ran MySQL using InnoDB for the table engine. For this benchmark, it is the fastest engine that provides similar durability to Stasis. We linked the benchmark's executable to the libmysqld daemon library, bypassing the IPC layer. Experiments that used IPC were orders of magnitude slower.

Figure 6 presents the performance of the three Stasis variants, and the Oasys plugins implemented on top of other systems. In this test, none of the systems were memory bound. As we can see, Stasis performs better than the baseline systems, which is not surprising, since it exploits the weaker durability requirements.

In non-memory bound systems, the optimizations nearly double Stasis' performance by reducing the CPU overhead of marshalling and unmarshalling objects, and by reducing the size of log entries written to disk.

To determine the effect of the optimization in memory bound systems, we decreased Stasis' page cache size, and used O_DIRECT to bypass the operating system's disk cache. We partitioned the set of objects so that 10% fit in a *hot set*. Figure 6 also presents Stasis' performance as we varied the percentage of object updates that manipulate the hot set. In the memory bound test, we see that update/flush indeed improves memory utilization.

5.4 Request reordering

We are interested in enabling Stasis to manipulate sequences of application requests. By translating these requests into logical operations (such as those used for logical undo), we can manipulate and optimize such requests. Because logical operations generally correspond to application-level operations, application developers can easily determine whether logical operations may be reordered, transformed, or even dropped from the stream of requests that Stasis is processing. For example, requests that manipulate disjoint sets of data can be split across many nodes, providing load balancing. Requests that update the same piece of information can be merged into a single request; RVM's "log merging" implements this type of optimization [42]. Stream aggregation techniques and relational algebra operators could be used to transform data efficiently while it is laid out sequentially in non-transactional memory.

To experiment with the potential of such optimizations, we implemented a single-node request-reordering scheme that increases request locality during a graph traversal. The graph traversal produces a sequence of read requests that are partitioned according to their physical location in the page file. Partition sizes are chosen to fit inside the buffer pool. Each partition is processed until there are no more outstanding requests to read from it. The process iterates until the traversal is complete.

We ran two experiments. Both stored a graph of fixedsize objects in the growable array implementation that is

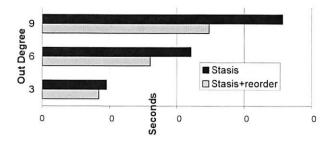


Figure 8: OO7 benchmark style graph traversal. The optimization performs well due to the presence of non-local nodes.

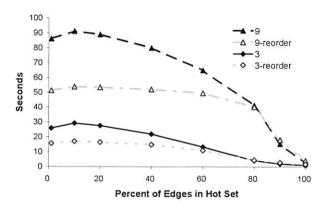


Figure 9: Hot-set based graph traversal for random graphs with out-degrees of 3 and 9. The multiplexer has low overhead, and improves performance when the graph has poor locality.

used as our linear hash table's bucket list. The first experiment (Figure 8) is loosely based on the OO7 database benchmark [9]. We hard-code the out-degree of each node and use a directed graph. Like OO7, we construct graphs by first connecting nodes together into a ring. We then randomly add edges until obtaining the desired out-degree. This structure ensures graph connectivity. Nodes are laid out in ring order on disk so at least one edge from each node is local.

The second experiment measures the effect of graph locality (Figure 9). Each node has a distinct hot set that includes the 10% of the nodes that are closest to it in ring order. The remaining nodes are in the cold set. We do not use ring edges for this test, so the graphs might not be connected. We use the same graphs for both systems.

When the graph has good locality, a normal depth-first search traversal and the prioritized traversal both perform well. As locality decreases, the partitioned traversal algorithm outperforms the naive traversal.

6 Related Work

6.1 Database Variations

This section discusses database systems with goals similar to ours. Although these projects were successful in many respects, each extends the range of a fixed abstract data model. In contrast, Stasis can support (in theory) any of these models and their extensions.

6.1.1 Extensible databases

Genesis is an early database toolkit that was explicitly structured in terms of the physical data models and conceptual mappings described above [5]. It allows database implementors to swap out implementations of the components defined by its framework. Like later systems (including Stasis), it supports custom operations.

Subsequent extensible database work builds upon these foundations. The Exodus [8] database toolkit is the successor to Genesis. It uses abstract data type definitions, access methods and cost models to generate query optimizers and execution engines automatically.

Object-oriented database systems [28] and relational databases with support for user-definable abstract data types (such as POSTGRES [52]) provide functionality similar to extensible database toolkits. In contrast to database toolkits, which leverage type information as the database server is compiled, object-oriented and object-relational databases allow types to be defined at runtime.

Both approaches extend a fixed high-level data model with new abstract data types. This is of limited use to applications that are not naturally structured in terms of queries over sets.

6.1.2 Modular databases

The database community is also aware of this gap. A recent survey [13] enumerates problems that plague users of state-of-the-art database systems. Essentially, it finds that modern databases are too complex to be implemented or understood as a monolithic entity. Instead, they have become unpredictable and unmanageable, preventing them from serving large-scale applications and small devices. Rather than concealing performance issues, SQL's declarative interface prevents developers from diagnosing and correcting underlying problems.

The study suggests that researchers and the industry adopt a highly modular "RISC" database architecture. This architecture would be similar to a database toolkit, but would standardize the interfaces of the toolkit's components. This would allow competition and specialization among module implementors, and distribute the effort required to build a full database [13].

Streaming applications face many of the problems that RISC databases could address. However, it is unclear whether a single interface or conceptual mapping would meet their needs. Based on experiences with their system, the authors of StreamBase argue that "one size fits all" database engines are no longer appropriate. Instead, they argue that the market will "fracture into a collection of independent...engines" [51]. This is in contrast to the RISC approach, which attempts to build a database in terms of interchangeable parts.

We agree with the motivations behind RISC databases and StreamBase, and believe they complement each other and Stasis well. However, our goal differs from these systems; we want to support applications that are a poor fit for database systems. As Stasis matures we hope that it will enable a wide range of transactional systems, including improved DBMSs.

6.2 Transactional Programming Models

Transactional programming environments provide semantic guarantees to the programs they support. To achieve this goal, they provide a single approach to concurrency and transactional storage. Therefore, they are complementary to our work; Stasis provides a substrate that makes it easier to implement such systems.

6.2.1 Nested Transactions

Nested transactions allow transactions to spawn subtransactions, forming a tree. Linear nesting restricts transactions to a single child. Closed nesting rolls children back when the parent aborts [37]. Open nesting allows children to commit even if the parent aborts.

Closed nesting uses database-style lock managers to allow concurrency within a transaction. It increases fault tolerance by isolating each child transaction from the others, and retrying failed transactions. (MapReduce is similar, but uses language constructs to statically enforce isolation [15].)

Open nesting provides concurrency between transactions. In some respect, nested top actions provide open, linear nesting, as the actions performed inside the nested top action are not rolled back when the parent aborts. (We believe that recent proposals to use open, linear nesting for software transactional memory will lead to a programming style similar to Stasis' [38].) However, logical undo gives the programmer the option to compensate for nested top actions. We expect that nested transactions could be implemented with Stasis.

6.2.2 Distributed Programming Models

Nested transactions simplify distributed systems; they isolate failures, manage concurrency, and provide dura-

bility. In fact, they were developed as part of Argus, a language for reliable distributed applications. An Argus program consists of guardians, which are essentially objects that encapsulate persistent and atomic data. Accesses to *atomic* data are serializable, while *persistent* data is atomic data that is stored on disk [29].

Originally, Argus only supported limited concurrency via total isolation, but was extended to support high concurrency data structures. Concurrent data structures are stored in non-atomic storage, but are augmented with information in atomic storage. This extra data tracks the status of each item stored in the structure. Conceptually, atomic storage used by a hash table would contain the values "Not present", "Committed" or "Aborted; Old Value = x" for each key in (or missing from) the hash. Before accessing the hash, the operation implementation would consult the appropriate piece of atomic data, and update the non-atomic data if necessary. Because the atomic data is protected by a lock manager, attempts to update the hash table are serializable. Therefore, clever use of atomic storage can be used to provide logical locking.

Efficiently tracking such state is not straightforward. For example, their hash table implementation uses a log structure to track the status of keys that have been touched by active transactions. Also, the hash table is responsible for setting policies regarding granularity and timing of disk writes [54]. Stasis operations avoid this complexity by providing logical undos, and by leaving lock management to higher-level code. This separates write-back and concurrency control policies from data structure implementations.

Camelot made a number of important contributions, both in system design, and in algorithms for distributed transactions [16]. It leaves locking to application level code, and updates data in place. (Argus uses shadow copies to provide atomic updates.) Camelot provides two logging modes: physical redo-only (no-steal, noforce) and physical undo/redo (steal, no-force). Because Camelot does not support logical undo, concurrent operations must be implemented similarly to those built with Argus. Camelot is similar to Stasis in that its low-level C interface is designed to enable multiple higher-level programming models, such as Avalon's C++ interface or an early version of RVM. However, like other distributed programming models, Camelot focuses on a particular class of distributed transactions. Therefore, it hard-codes assumptions regarding the structure of nested transactions, consensus algorithms, communication mechanisms, and so on.

More recent transactional programming schemes allow for multiple transaction implementations to cooperate as part of the same distributed transaction. For example, X/Open DTP provides a standard networking proto-

col that allows multiple transactional systems to be controlled by a single transaction manager [53]. Enterprise Java Beans is a standard for developing transactional middleware on top of heterogeneous storage. Its transactions may not be nested. This simplifies its semantics, and leads to many, short transactions, improving concurrency. However, flat transactions are somewhat rigid, and lead to situations where committed transactions have to be manually rolled back by other transactions [47]. The Open Multithreaded Transactions model is based on nested transactions, incorporates exception handling, and allows parents to execute concurrently with their children [24].

QuickSilver is a distributed transactional operating system. It provides a transactional IPC mechanism, and allows varying degrees of isolation, both to support legacy code, and to provide an appropriate environment for custom transactional software [21]. By providing an environment that allows multiple, independently written, transactional systems to interoperate, QuickSilver would complement Stasis nicely.

The QuickSilver project showed that transactions can meet the demands of most applications, provided that long-running transactions do not exhaust system resources, and that flexible concurrency control policies are available. Nested transactions are particularly useful when a series of program invocations form a larger logical unit [43].

Clouds is an object-oriented, distributed transactional operating system. It uses shared abstract types [44] and per-object atomicity specifications to provide concurrency control among the objects in the system [2]. These formalisms could be used during the design of high-concurrency Stasis operations.

6.3 Data Structure Frameworks

As mentioned in Sections 2.2 and 5, Berkeley DB is a system quite similar to Stasis, and gives application programmers raw access to transactional data structures such as a single-node B-Tree and hash table [45].

Cluster hash tables provide a scalable, replicated hash table implementation by partitioning the table's buckets across multiple systems [20]. Boxwood treats each system in a cluster of machines as a "chunk store," and builds a transactional, fault tolerant B-Tree on top of the chunks that these machines export [32].

Stasis is complementary to Boxwood and cluster hash tables; those systems intelligently compose a set of systems for scalability and fault tolerance. In contrast, Stasis makes it easy to push intelligence into the individual nodes, allowing them to provide primitives that are appropriate for the higher-level service.

6.4 Data layout policies

Data layout policies make decisions based upon assumptions about the application. Ideally, Stasis would allow application-specific layout policies to be used interchangeably, This section describes strategies for data layout that we believe Stasis could eventually support.

Some large object storage systems allow arbitrary insertion and deletion of bytes [10] within the object, while typical file systems provide append-only allocation [34]. Record-oriented allocation, such as in VMS Record Management Services [39] and GFS [18], breaks files into addressable units. Write-optimized file systems lay files out in the order they were written rather than in logically sequential order [41].

Schemes to improve locality among small objects exist as well. Relational databases allow users to specify the order in which tuples will be laid out, and often leave portions of pages unallocated to reduce fragmentation as new records are allocated.

Memory allocation routines such as Hoard [7] and McRT-malloc [23] address this problem by grouping allocated data by thread or transaction, respectively. This increases locality, and reduces contention created by unrelated objects stored in the same location. Stasis' current record allocator is based on these ideas (Section 3.5).

Allocation of records that must fit within pages and be persisted to disk raises concerns regarding locality and page layouts. Depending on the application, data may be arranged based upon hints [46], pointer values and write order [31], data type [25], or access patterns [56].

We are interested in allowing applications to store records in the transaction log. Assuming log fragmentation is kept to a minimum, this is particularly attractive on a single disk system. We plan to use ideas from LFS [41] and POSTGRES [52] to implement this.

7 Future Work

Complexity problems may begin to arise as we attempt to implement more extensions to Stasis. However, Stasis' implementation is still fairly simple:

- The core of Stasis is roughly 3000 lines of C code, and implements the buffer manager, IO, recovery, and other systems.
- Custom operations account for another 3000 lines.
- Page layouts and logging implementations account for 1600 lines.

The complexity of the core of Stasis is our primary concern, as it contains the hard-coded policies and assumptions. Over time, it has shrunk as functionality has moved into extensions. We expect this trend to continue as development progresses.

A resource manager is a common pattern in system software design, and manages dependencies and ordering constraints among sets of components. Over time, we hope to shrink Stasis' core to the point where it is simply a resource manager that coordinates interchangeable implementations of the other components.

8 Conclusion

We presented Stasis, a transactional storage library that addresses the needs of system developers. Stasis provides more opportunities for specialization than existing systems. The effort required to extend Stasis to support a new type of system is reasonable, especially when compared to current practices, such as working around limitations of existing systems, breaking guarantees regarding data integrity, or reimplementing the entire storage infrastructure from scratch.

We demonstrated that Stasis provides fully concurrent, high-performance transactions, and explored how it can support a number of systems that currently make use of suboptimal or ad-hoc storage approaches. Finally, we described how Stasis can be extended in the future to support a larger range of systems.

9 Acknowledgements

Thanks to shepherd Bill Weihl for helping us present these ideas well, or at least better. The idea behind the Oasys buffer manager optimization is from Mike Demmer; he and Bowei Du implemented Oasys. Gilad Arnold and Amir Kamil implemented pobj. Jim Blomo, Jason Bayer, and Jimmy Kittiyachavalit worked on an early version of Stasis.

Thanks to C. Mohan for pointing out that per-object LSNs may be inadvertently overwritten during recovery. Jim Gray suggested we use a resource manager to track dependencies within Stasis and provided feedback on the LSN-free recovery algorithms. Joe Hellerstein and Mike Franklin provided us with invaluable feedback.

Portions of this work were performed at Intel Research Berkeley.

10 Availability

Additional information, and Stasis' source code is available at:

http://www.cs.berkeley.edu/~sears/stasis/

References

- AGRAWAL, R., CAREY, M. J., AND LIVNY, M. Concurrency control performance modeling: Alternatives and implications. ACM Transactions on Database Systems (1987).
- [2] ALLCHIN, J. E., AND MCKENDRY, M. S. Synchronization and recovery of actions. In *PODC* (1983), pp. 31–44.
- [3] ASTRAHAN, M. ET AL. System R: Relational approach to database management. ACM Transactions on Database Systems 1, 2 (1976), 97–137.
- [4] BATORY, D. S. Conceptual-to-internal mappings in commercial database systems. In *PODS* (1984), pp. 70–78.
- [5] BATORY, D. S., BARNETT, J. R., GARZA, J. F., SMITH, K. P., TSUKUDA, K., TWICHELL, B. C., AND WISE, T. E. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering* 14, 11 (November 1988), 1711–1729.
- [6] BATORY, D. S., AND GOTLIEB, C. C. A unifying model of physical databases. ACM Transactions on Database Systems 7, 4 (1982), 509–539.
- [7] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. ACM SIGPLAN Notices 35, 11 (2000), 117–128.
- [8] CAREY, M. J., DEWITT, D. J., FRANK, D., GRAEFE, G., MU-RALIKRISHNA, M., RICHARDSON, J., AND SHEKITA, E. J. The architecture of the EXODUS extensible DBMS. In OODS (1986), pp. 52–65.
- [9] CAREY, M. J., DEWITT, D. J., AND NAUGHTON, J. F. The OO7 benchmark. In SIGMOD (1993), pp. 12–21.
- [10] CAREY, M. J., DEWITT, D. J., RICHARDSON, J. E., AND SHEKITA, E. J. Object and file management in the EXODUS extensible database system. In VLDB (1986), pp. 91–100.
- [11] CHANDRASEKARAN, S., AND FRANKLIN, M. Streaming queries over streaming data. In VLDB (2002).
- [12] CHAUDHURI, S., AND DAYAL, U. An overview of data ware-housing and OLAP technology. ACM SIGMOD Record 26, 1 (1997), 65–74.
- [13] CHAUDHURI, S., AND WEIKUM, G. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In VLDB (2000).
- [14] CODD, E. F. A relational model of data for large shared data banks. Communications of the ACM 13, 6 (June 1970), 377–387.
- [15] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In OSDI (2004).
- [16] EPPINGER, J. L., MUMMERT, L. B., AND SPECTOR, A. Z., Eds. Camelot and Avalon: A Distributed Transaction Facility. Morgan Kaufmann, 1991.
- [17] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. Cluster-based scalable network services. In SOSP (1997), pp. 78–91.
- [18] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The Google file system. In SOSP (2003), pp. 29–43.
- [19] GRAY, J., AND REUTERS, A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.
- [20] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. E. Scalable, distributed data structures for Internet service construction. In OSDI (2000), pp. 319–332.
- [21] HASKIN, R. L., MALACHI, Y., SAWDON, W., AND CHAN, G. Recovery management in QuickSilver. ACM Transactions on Computer Systems 6, 1 (1988), 82–108.

- [22] Hibernate. http://www.hibernate.org/.
- [23] HUDSON, R. L., SAHA, B., ADL-TABATABAI, A.-R., AND HERTZBERG, B. McRT-Malloc: A scalable transactional memory allocator. In *ISMM* (2006), pp. 74–83.
- [24] KIENZLE, J., STROHMEIER, A., AND ROMANOVSKY, A. B. Open multithreaded transactions: Keeping threads and exceptions under control. In WORDS (2001), pp. 197–205.
- [25] KIM, W., GARZA, J. F., BALLOU, N., AND WOELK, D. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering* (1990).
- [26] KOLODNER, E. K., AND WEIHL, W. E. Atomic incremental garbage collection and recovery for a large stable heap. In SIG-MOD (1993), pp. 177–186.
- [27] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. ACM Transactions on Database Systems 6, 2 (June 1981), 213–226.
- [28] LAMB, C., LANDIS, G., ORENSTEIN, J. A., AND WEINREB, D. The ObjectStore database system. *Communications of the ACM 34*, 10 (1991), 50–63.
- [29] LISKOV, B. Distributed programming in Argus. Communications of the ACM 31, 3 (March 1988), 300–312.
- [30] LITWIN, W. Linear hashing: A new tool for file and table addressing. In VLDB (1980), pp. 224–232.
- [31] LOHMAN, G., LINDSAY, B., PIRAHESH, H., AND SCHIEFER, K. B. Extensions to Starburst: Objects, types, functions, and rules. *Communications of the ACM 34*, 10 (October 1991), 95– 109.
- [32] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In OSDI (2004), pp. 105–120.
- [33] MCHUGH, J., ABITEBOUL, S., GOLDMAN, R., QUASS, D., AND WIDOM, J. Lore: A database management system for semistructured data. ACM SIGMOD Record 26, 3 (September 1997), 54–66.
- [34] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. ACM Transactions on Computer Systems (1984).
- [35] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. M. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems 17, 1 (1992), 94–162.
- [36] MOHAN, C., AND LEVINE, F. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. ACM Press, 1992.
- [37] MOSS, J. E. B. Nested transactions: An approach to reliable distributed computing. MIT, 1985.
- [38] Moss, J. E. B. Open nested transactions: Semantics and support. In WMPI (2006).
- [39] OpenVMS Record Management Services Reference Manual, June 2002.
- [40] REISER, H. T. ReiserFS. http://www.namesys.com.
- [41] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In SOSP (1992).
- [42] SATYANARAYANAN, M., MASHBURN, H. H., KUMAR, P., STEERE, D. C., AND KISTLER, J. J. Lightweight recoverable virtual memory. ACM Transactions on Computer Systems 12, 1 (February 1994), 33–57.
- [43] SCHMUCK, F. B., AND WYLLIE, J. C. Experience with transactions in QuickSilver. In SOSP (1991), pp. 239–253.

- [44] SCHWARZ, P. M., AND SPECTOR, A. Z. Synchronizing shared abstract types. ACM Transactions on Computer Systems 2, 3 (April 1984), 223–250.
- [45] SELTZER, M., AND OLSEN, M. LIBTP: Portable, modular transactions for UNIX. In *Usenix* (January 1992).
- [46] SHEKITA, E. J., AND ZWILLING, M. J. Cricket: A mapped, persistent object store. In POS (1990), pp. 89–102.
- 47] SILAGHI, R., AND STROHMEIER, A. Critical evaluation of the EJB transaction model. In *FIDJI* (2002), pp. 15–28.
- [48] STEIN, L. How Perl saved the Human Genome Project. Dr Dobb's Journal (July 2001).
- [49] STONEBRAKER, M. Operating system support for database management. Communications of the ACM 24, 7 (July 1981), 412– 418
- [50] STONEBRAKER, M. Inclusion of new types in relational data base systems. In *ICDE* (1986), pp. 262–269.
- [51] STONEBRAKER, M., AND ÇETINTEMEL, U. "One size fits all": An idea whose time has come and gone. In *ICDE* (2005), pp. 2–11.
- [52] STONEBRAKER, M., AND KEMNITZ, G. The POSTGRES nextgeneration database management system. *Communications of the* ACM 34, 10 (October 1991), 79–92.
- [53] THE OPEN GROUP. Distributed Transaction Processing: Reference Model, 1996.
- [54] WEIHL, W., AND LISKOV, B. Implementation of resilient, atomic data types. ACM Transactions on Programming Languages and Systems 7, 2 (April 1985), 244–269.
- [55] WEIKUM, G., AND SCHEK, H.-J. Architectural issues of transaction management in multi-layered systems. In VLDB (1984), pp. 454–465.
- [56] YONG, V.-F., NAUGHTON, J. F., AND YU, J.-B. Storage reclamation and reorganization in client-server persistent object stores. In *ICDE* (1994), pp. 120–131.
- [57] ZEEBERG, B. R., RISS, J., KANE, D. W., BUSSEY, K. J., UCHIO, E., LINEHANN, W. M., BARRETT, J. C., AND WEIN-STEIN, J. N. Mistaken identifiers: Gene name errors can be introduced inadvertently when using Excel in bioinformatics. *BMC Bioinformatics* (2004).
- [58] ZHAO, Y., DESHPANDE, P. M., AND NAUGHTON, J. F. An array-based algorithm for simultaneous multidimensional aggregates. In SIGMOD (1997), pp. 159–170.

Notes

¹The "A" in ACID really means "atomic persistence of data," rather than "atomic in-memory updates," as the term is normally used in systems work; the latter is covered by "C" and "I" [19].

²For efficiency, undo and redo operations are packed into a single log entry. Both must take the same parameters.

³We found that the relative performance of Berkeley DB and Stasis under single-threaded testing is sensitive to file system choice, and we plan to investigate the reasons why the performance of Stasis under ext3 is degraded. However, the results relating to the Stasis optimizations are consistent across file system types.

⁴Multi-threaded benchmarks were performed using an ext3 file system. Concurrency caused both Berkeley DB and Stasis to behave unpredictably under ReiserFS. Stasis' multi-threaded throughput was significantly better than Berkeley DB's with both file systems.

⁵These figures do not include the simple LSN-free object logic required for recovery, as Stasis does not yet support LSN-free operations.

SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques

Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak

Rob Ennals

University of California, Berkeley

Intel Research Berkeley

{zf,jcondit,zra,ibagrak}@cs.berkeley.edu

robert.ennals@intel.com

Matthew Harren, George Necula, Eric Brewer

University of California, Berkeley {matth, necula, brewer}@cs.berkeley.edu

We present SafeDrive, a system for detecting and recovering from type safety violations in software extensions. SafeDrive has low overhead and requires minimal changes to existing source code. To achieve this result, SafeDrive uses a novel type system that provides finegrained isolation for existing extensions written in C. In addition, SafeDrive tracks invariants using simple wrappers for the host system API and restores them when recovering from a violation. This approach achieves finegrained memory error detection and recovery with few code changes and at a significantly lower performance cost than existing solutions based on hardware-enforced domains, such as Nooks [33], L4 [21], and Xen [13], or software-enforced domains, such as SFI [35]. The principles used in SafeDrive can be applied to any large system with loadable, error-prone extension modules.

In this paper we describe our experience using SafeDrive for protection and recovery of a variety of Linux device drivers. In order to apply SafeDrive to these device drivers, we had to change less than 4% of the source code. SafeDrive recovered from all 44 crashes due to injected faults in a network card driver. In experiments with 6 different drivers, we observed increases in kernel CPU utilization of 4–23% with no noticeable degradation in end-to-end performance.

1 Introduction

Large systems such as operating systems and web servers often provide an extensibility mechanism that allows the behavior of the system to be customized for a particular usage scenario. For example, device drivers adapt the behavior of an operating system to a particular hardware configuration, and web server modules adapt the behavior of the web server to the content or performance needs

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0509544.

of a particular web site. However, such extensions are often responsible for a disproportionately large number of bugs in the system [9, 33], and bugs in an extension can often cause the entire system to fail. Our goal is to improve the reliability of extensible systems without requiring significant changes to the core of the system. To do so, we must *isolate* existing extensions, preferably with little modification, *restore system invariants* when they fail, *restart* them automatically for availability, and (ideally) *restore active sessions*.

In this paper, we focus on the specific problem of improving device driver reliability. Previous systems have attempted to address this problem using some form of lightweight protection domain for extensions. For example, the Nooks project [32, 33] runs Linux device drivers in an isolated portion of the kernel address space, modifying kernel API calls to move data into and out of the extension. This approach prevents drivers from overwriting kernel memory at the cost of relatively expensive driver/kernel boundary crossings.

Our system, SafeDrive, takes a different approach to improving extension reliability. Instead of using hardware to enforce isolation, SafeDrive uses language-based techniques similar to those used in type-safe languages such as Java. Specifically, SafeDrive adds type-based checking and restart capabilities to *existing* device drivers written in C without hardware support or major OS changes (i.e., without adding a new protection domain mechanism). We have four primary goals for SafeDrive:

Fine-grained type-based isolation: We detect
memory and type errors on a per-pointer basis,
whereas previous work has only attempted to provide per-extension memory safety. SafeDrive ensures that data of the correct type is used in
kernel API calls and in shared data structures.
This advantage is critical, because it means that
SafeDrive can catch memory and type violations be-

fore they corrupt data, even for violations that occur entirely within the driver. Thus, we can prevent the kernel or devices from receiving incorrect data for these cases. SafeDrive can also catch more memory-related bugs than hardware-based approaches; specifically, SafeDrive can catch errors that violate type safety but do not trigger VM faults. In addition, because errors are caught as they occur, SafeDrive can provide fine-grained error reports for debugging.

- Lower overhead for isolation: SafeDrive exhibits lower overhead in general, particularly for extensions with many crossings (for which Nooks admits it is a poor fit [33, Sec. 6.4]). Compared with SafeDrive, hardware-enforced isolation incurs additional overhead due to domain changes, page table updates, and data copying. Also, the stronger type invariants that SafeDrive maintains makes it possible to check many pointer operations statically.
- Non-intrusive evolutionary design. SafeDrive provides type safety without changing the structure of the host system (e.g., the OS kernel) significantly and without rewriting extensions to use a new language or API.
- Protection against buggy (but not malicious) extensions. In rare cases where true type safety would require significant changes to the extension or to the host API, we prefer to trust individual operations whose safety we cannot verify and gradually migrate to more complete isolation over time. For example, our current implementation does not yet attempt to verify memory allocation, deallocation, and mapping operations. In addition, we make no attempt to protect the system from extensions that abuse CPU, memory, or other resources (unlike OKE [5] and Singularity [27]). As a consequence, SafeDrive is able to guard against mistakes made by the author of the extension but does not attempt to protect against a malicious adversary capable of exploiting specific behaviors of our system.

C and its variants have a number of important constructs that can be used to cause violations. In addition to the most obvious issue of out-of-bounds array accesses, C also has fundamental problems due to unions, null-terminated strings, and other constructs. To transform a driver written in C (which includes all Linux drivers) into one that obeys stricter type safety requirements, we must fix all of these flaws without requiring extensive rewrites and ideally without requiring modifications to the kernel.

The existing approaches to type safety for C involve the use of "fat" pointers, which contain both the pointer and its bounds information. CCured [24], for example, can make a legacy C program memory-safe by converting most of its pointers into fat pointers and then inserting run-time checks to enforce bounds constraints. However, this approach is not realistic for drivers or for the kernel, since it modifies the layout of every structure containing pointers as well as every kernel API function that uses pointers. To use CCured effectively in this context, we would need to "cure" the entire kernel and all of its drivers together, which is impractical.

Instead, SafeDrive uses a novel type system for pointers, called Deputy, that can enforce memory safety for most programs without resorting to the use of fat pointers and thus without requiring changes to the layout of data or the driver API. The key insight is that most of the required pointer bounds information is already present in the driver code or in the API—just not in a form that the compiler currently understands. Deputy uses type annotations, particularly in header files for APIs and shared structures, to identify this information in places where it already exists.

Although adding annotations to kernel headers or driver code may seem like a burden, there are many reasons why this approach is a practical one. First, the required annotations are typically very simple, allowing programmers to easily express known relationships between variables and fields (e.g., "p points to an array of length n"). Second, the cost of annotating kernel headers is a one-time cost; once the headers are annotated, the marginal cost of annotating additional drivers is much smaller. Third, annotations are only mandatory for the driver-kernel interface, while the unannotated data structures internal to the driver can use fat pointers. About 600 Deputy annotations were added to kernel headers for the 6 drivers we tested (Section 5.2).

Any solution based on run-time enforcement of isolation must provide a mechanism for dealing with violations. In SafeDrive we assume that extensions are restartable provided that certain system invariants are restored. In the case of Linux device drivers, invariant restoration consists of releasing a number of resources allocated by the driver and unregistering any name space entries registered by the driver (e.g., new device entries or file system entries), both of which we will refer to as *updates*. In order to undo updates during fault recovery, we track them using wrappers for the relevant API calls. Because SafeDrive allows an extension to operate safely in the kernel address space without the use of a hardware-enforced protection domain, the task of managing and recovering kernel resources is greatly simplified.

As with Nooks, SafeDrive cannot prevent every extension-related crash, nor can it guarantee that malicious code cannot abuse the machine or the device. However, because SafeDrive can catch errors that corrupt the driver itself without corrupting other parts of the system, we expect it to catch more errors, and we expect it to

46

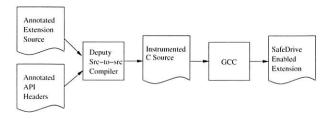


Figure 1: Compilation of an extension in SafeDrive.

catch them earlier.

We have implemented SafeDrive for Linux device drivers, and we have used the system on network drivers, sound drivers, and video drivers, among others. Our experiments indicate that SafeDrive provides safety and recovery with significantly less run-time overhead than Nooks, especially when many calls to/from the driver are made (e.g., 12% vs. 111% overhead in one benchmark, and 4% vs. 46% in another). The main conclusion to draw from these experiments is that language-based techniques can provide fine-grained error detection at significantly lower cost by eliminating the need for expensive kernel/extension boundary crossings.

In Section 2, we present an overview of the SafeDrive system, including the compile-time and run-time components. Then, in Section 3 and Section 4, we describe in detail the required annotations and the implementation of the recovery system. Section 5 describes our experiments. Finally, Section 6 and Section 7 discuss related work and our conclusions.

2 SafeDrive Overview

In SafeDrive, isolated recoverable extensions require support from the programmer, the compiler, and the runtime system. The programmer is responsible for inserting type annotations that describe pointer bounds, and the compiler uses these annotations to insert appropriate run-time checks. The runtime system contains the implementation of the recovery subsystem.

The compiler is implemented as a source-to-source transformation. Given the annotated C code, the Deputy source-to-source compiler produces an instrumented version of this source code that contains the appropriate checks. This instrumented code is then compiled with GCC. Most of the annotations required for this process are found in the system header files, where they provide bounds information for API calls in addition to the recovery system interface. The compilation process is illustrated in Figure 1.

At run time, a SafeDrive-enabled extension is loaded into the same address space as the host system and is linked to both the host system and the SafeDrive runtime system. Because all code runs in the same address space,

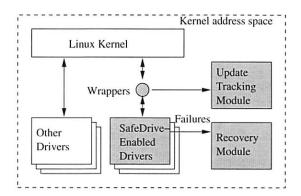


Figure 2: Block diagram of SafeDrive for Linux device drivers. Gray boxes indicate new or changed components.

no special handling of calls between the host system and the extension is required, apart from the run-time checks that the Deputy compiler inserts based on each function's annotations. The extension can read and write shared data structures directly, again using the Deputy-inserted run-time checks to verify the safety of these operations. Note that the host system does not need to be annotated or compiled with the Deputy compiler. Another consequence of this binary compatibility property is that non-SafeDrive extensions can be used with a SafeDrive-enabled host system as-is.

The SafeDrive runtime system is responsible for "update tracking" and recovery. It maintains data structures tracking the list of updates the extension has done to kernel state. Whenever a Deputy check fails, the control is passed to the SafeDrive runtime system, which attempts to cancel out these updates and restart the failed extension. Figure 2 shows the structure of the SafeDrive runtime system for the Linux kernel.

Although this paper focuses on Linux device drivers, we believe that the principles used in SafeDrive could be applied to a wide range of other extensible systems without a large amount of additional effort.

2.1 Example

Figure 3 shows some sample code adapted from a Linux device driver. Here the programmer has added the Deputy annotation "count (info_count)" to the buffer_info field, which indicates that info_count stores the number of elements in this array. The original code contained the same information in comments only; thus, the existence of this relationship between structure fields was previously hidden to the compiler.

Code in boldface shows run-time checks that SafeDrive inserts into the instrumented version of the file (see Figure 1) to enforce isolation and to support

```
struct e1000_tx_ring {
  unsigned int info-count;
  struct e1000_buffer * count(info_count)
      buffer_info;
};
static boolean_t
e1000_clean_tx_irg(
    struct e1000_adapter *adapter,
    struct e1000_tx_ring *tx_ring)
  assert(tx_ring != NULL);
  spin_lock(&tx_ring->tx_lock);
  track_spin_lock(&tx_ring->tx_lock);
  i = tx_ring->next_to_clean;
  assert(0 \le i \&\& i \le tx\_ring->info\_count);
  eop = tx_ring->buffer_info[i]
                      .next_to_watch;
  track_spin_unlock(&tx_ring->tx_lock);
  spin_unlock(&tx_ring->tx_lock);
```

Figure 3: Example adapted from the Linux e1000 network card driver. The one programmer-inserted annotation is underlined. The boldface code shows Deputy run-time checks that are inserted into the instrumented version of the file.

recovery. The first and third checks are assertions that enforce memory safety. The first check ensures that tx_ring is non-null, which is required because Deputy assumes that unannotated function arguments are either null or point to a single element of the declared type, much like references in a type-safe language. The third check in the example enforces the bounds declared for the buffer_info field before it is accessed. Note that a simple optimization ensures that we do not insert redundant checks every time tx_ring is dereferenced; indeed, the low overhead of SafeDrive is in part due to the fact that most pointer accesses require very simple checks or no checks at all.

If either of these assertions fail, SafeDrive invokes the recovery subsystem. To support recovery, SafeDrive inserts code to track the invariants that must be restored, as seen in the second and fourth boldface statements in this example.

This example shows that SafeDrive preserves the binary interface with the rest of the kernel, inserts relatively few checks (tracked resource allocation is rare, and most pointers point to a single object), at the expense of a few annotations that capture simple invariants. In the next

two sections, we describe in detail our type system and the associated recovery system.

3 The Deputy Type System

The goal of the Deputy type system is to prevent pointer bounds errors through a combination of compile-time and run-time checking. Adding these checks is a challenging task because the C language provides no simple way to determine at run time whether a pointer points to an allocated object of the appropriate type. Previous systems, such as CCured [24], addressed this problem by replacing pointers with multi-word pointers, known as "fat" pointers, that indicate the bounds for the pointer in question. Unfortunately, this approach changes the layout of data in the program, making it very difficult to "cure" one module or extension independently of the rest of the system.

In contrast, Deputy's types allow the programmer to specify pointer bounds in terms of other variables or structure fields in the program. Deputy's annotations also allow the programmer to identify null-terminated arrays and tagged unions. These annotations are flexible enough to accommodate a wide range of existing strategies for tracking pointer bounds. The key insight is that most pointers' bounds are present in the program in some form—just not a form that is obvious to the compiler. By adding appropriate annotations, we allow the compiler to insert memory safety checks throughout the program without changing the layout of the program's data structures.

Deputy is implemented as a source-to-source transformation that runs immediately after preprocessing. It is built on top of the CIL compiler framework [25], and the current prototype contains about 20,000 lines of OCaml code. Deputy has three phases:

- Inference. First, Deputy infers bounds annotations
 for every unannotated pointer in the program. For
 unannotated local variables, Deputy inserts an annotation that refers to new local variables that explicitly track the unannotated pointer's bounds; this
 approach is essentially a variant of the fat pointer
 approach. For globals, structure fields, and function parameters and results, Deputy assumes a default pointer type. We use the inference module
 from CCured [24] to improve local variable annotations and to identify global variables that may require manual annotation.
- Type Checking. Once all pointers have Deputy annotations, Deputy checks the code itself. It emits errors and run-time checks where appropriate.
- 3. *Optimization*. Since the type checking phase generates a large number of run-time checks, a Deputy-

specific optimization phase is used to eliminate checks that will never fail at run-time and to identify checks that will definitely fail at run time.

In the remainder of this section, we will discuss Deputy's types and run-time checks in more detail.

3.1 Deputy Type Annotations

In this section, we discuss Deputy's type annotations and their associated run-time checks. These type annotations allow Deputy to verify code that uses unsafe C features, such as pointer arithmetic, null-terminated strings, and union types. We focus on the informal semantics of these annotations; the technical details of the Deputy type system and its soundness argument are beyond the scope of this paper.

The annotations described in this section were chosen because they represent a reasonably large range of common C programming practices. They are simple enough to allow the type system to reason about them effectively, and yet they are expressive enough to be usable in real-world C programs.

It is important to note that the type annotations described in this section are *not* trusted by the compiler. Deputy checks when assigning a value to a variable that the value has the appropriate type; when using a variable, the type checker assumes that it contains a value of the declared type. In other words, these annotations function much like the underlying C types. Of course, Deputy only checks one compilation unit at a time, and therefore it must assume that other compilation units adhere to the restrictions of any Deputy annotations on global variables and functions, even if those other compilation units are not compiled by Deputy.

Buffers. Deputy allows the user to specify the bounds of a pointer by using one of four type annotations: safe, sentinel, count(n), and bound(lo, hi). In these annotations, n, lo, and hi stand for expressions that can refer to other variable or field names in the immediately enclosing scope. For example, annotations on local variables can refer to other local variables in the same function, and annotations on structure fields can refer to other fields of the same structure. These annotations can be written after any pointer type in the program; for example, a variable named buf could be declared with the syntax int * count(len) buf, which means that the variable len holds the number of elements in buf. The meanings of these annotations are as follows:

 The safe annotation indicates that the pointer is either null or points to a single element of the base type. Such pointers are the most common kind of

- pointer in C programs, and they typically require only a null check at dereference.
- The sentinel annotation indicates that a pointer is useful only for comparisons and not for dereference. This annotation is typically used for pointers that point immediately after an allocated area, as permitted by the ANSI C standard.
- The count (n) annotation indicates that the pointer is either null or points to an array of at least n elements. When accessing an element of this array, Deputy will insert a run-time check verifying that the pointer is non-null and the index is between zero and n.
- The bound(10, hi) annotation indicates that the pointer is either null or points into an array with lower and upper bounds 10 and hi. When accessing this array or applying pointer arithmetic to this pointer, Deputy will verify at run time that the pointer remains within the stated bounds.

Deputy allows casts between pointers with these annotations, inserting run-time checks as appropriate. For example, when casting a count (n) pointer to a safe pointer, Deputy will check that $n \ge 1$. Similarly, when casting a pointer p with annotation count (n) to a pointer with annotation bound (lo, hi), Deputy will verify that lo \le p and that $p + n \le$ hi.

The fourth annotation, bound (lo, hi), is quite general and can be used to describe a wide range of pointer bound invariants. For example, if p points to an array whose upper bound is given by a sentinel pointer e, we could annotate p with bound (p, e), which indicates that p points to an area bounded by itself and e. In fact, the sentinel, safe, and count (n) annotations are actually special cases of bound (lo, hi); when used on a pointer named p, they are equivalent to bound (p, p), bound (p, p + 1), and bound (p, p + n), respectively. Thus, when checking these annotations, it suffices to consider the bound (lo, hi) annotation alone.

The invariant maintained by Deputy is as follows. At run time, any pointer whose type is annotated with bound (10, hi) must either be null or have a value between 10 and hi, inclusive. That is, for a pointer p of this type, we require that $p = 0 \mid \mid (10 \le p)$ && $p \le hi$. (Note that ANSI C allows p = hi as long as p is not dereferenced.) Furthermore, all of these pointers must be aligned properly with respect to the base type of this pointer. Given this invariant, we can verify the safety of a dereference operation by checking that p != 0 && $p \le hi$. In order to ensure that this invariant is maintained throughout the program's execution, Deputy inserts run-time checks before any operation that could potentially break this invariant, which in-

Figure 4: Code showing the usage of Deputy bounds annotations. Underlined code indicates programmer-inserted annotations; boldface code indicates checks performed by Deputy at run time.

cludes changing the pointer itself or any other variable that appears in 10 or hi. Since this process generates a large number of run-time checks, many of which are trivial (e.g., p <= p), Deputy provides an optimizer that is specifically designed to remove the statically verifiable checks that were generated by this process.

Figure 4 presents an example of Deputy-annotated code. This function finds and returns a pointer to the first null element in an array, if one exists. This example shows several annotations at work. The first argument, buf, is annotated with count (len), which indicates that len stores the length of this buffer. The return type of this function, int * safe, indicates that we return a pointer to a single element (or null). (This annotation is not strictly necessary since safe is the default annotation on most unannotated pointers.) In the body of the function, we use a sentinel type for end, indicating that it cannot be dereferenced. Also, we use cur and end for the bounds of cur, which allows us to increment cur until it reaches end without breaking cur's bounds invariant.

The boldface code in Figure 4 indicates the checks that were inserted automatically by Deputy based on the programmer-supplied annotations. When applying arithmetic to buf, we verify that buf is not null, since Deputy disallows arithmetic on null pointers. When dereferencing, incrementing, or returning cur, we verify that there is at least one element remaining in the array, which is a requirement for all of these operations. In many cases, Deputy's optimizer has removed checks that it determined to be unnecessary. For example, the result of buf + len is required to stay in bounds, so Deputy checks that buf <= buf + len <= buf + len; however, since len is known to be the length of buf, Deputy's optimizer has removed this check. In addition, the cur++ operation requires the same check as the dereference in the previous statement; in this exam-

Figure 5: Example of Deputy's nullterm annotation.

ple, Deputy's optimizer has removed the duplicate check. Note that there are still many opportunities for further optimization of these checks; for example, a flow-sensitive optimizer could eliminate the second assertion entirely. We leave such improvements to future work, though it is worth noting that improvements in Deputy's run-time overhead are still well within reach.

Null termination. In addition to the basic pointer bounds annotations, Deputy also provides a nullterm annotation that can be used in conjunction with any one of the above annotations. This annotation indicates that the elements beyond the upper bound described by the bound annotation are a null-terminated sequence; that is, the bound annotation describes a subset of a larger null-terminated sequence. For example, count (5) nullterm indicates that a pointer points to an array of five elements followed by a null-terminated sequence. Note that count (0) nullterm indicates that a pointer points to an empty array followed by a null-terminated sequence—that is, it is a standard nullterminated sequence.

To see why this annotation is useful, consider the declaration for the strlcpy() function shown in Figure 5. The argument dst is annotated as char * count(size-1) nullterm, meaning that it has an least size-1 bytes of real data followed by a nullterminated area (typically just a zero byte). The src argument is const char * count(0) nullterm, which just means that it is a standard null-terminated string with unknown minimum length. These annotations demonstrate the flexibility of the nullterm annotation, since Deputy is capable of describing both a standard null-terminated pointer as well as a null-terminated array with some known minimum size.

The checks inserted for null-terminated sequences are a straightforward extension of the checks for bounds annotations. For example, when dereferencing a null-terminated pointer, we verify only that the pointer is non-null. When incrementing a null-terminated pointer, we check that the pointer stays within the declared bounds, and if not, we check that it is not incremented past the null element. Note that the nullterm annotation can be safely dropped during a cast, but it cannot be safely added to a pointer that is not null-terminated.

```
struct e1000_option {
  enum {range_option, list_option} type;
  union {
    struct {
        int min, max;
    } range when(type == range_option);
    struct {
        int nr;
        struct e1000_opt_list {...} *p;
    } list when(type == list_option);
} arg;
};
```

Figure 6: Code showing usage of Deputy's when annotation, adapted from Linux's e1000 driver.

Tagged unions. Deputy also provides support for tagged unions. From the perspective of memory safety, C unions are a form of cast, allowing data of one type to be reinterpreted as data of another type if used improperly. C programmers usually use a tag field in the enclosing structure to determine which field of the union is currently in use, but the compiler cannot verify proper use of this tag. As with pointer bounds, Deputy allows the programmer to declare the conditions under which each field of the union is used so that these conditions can be verified at run time when one of the union's fields is accessed. To do so, the programmer adds the annotation "when (p)" to each field of the union, where p is a predicate that can refer to variable or field names in the immediately enclosing scope, and can use arbitrary side-effect-free arithmetic and logical operators.

For example, Figure 6 shows the annotations used by the Linux device driver e1000. In this example, the type field indicates which field of the union is currently selected. When using Deputy, the programmer can place the when annotations to indicate the correspondence between the tag and the choice of union field so that it can be checked when the union is accessed. For example, when reading the field range, Deputy will check that type == range_option. When the user wishes to change the tag field (or any field on which the when predicates depend), Deputy verifies that pointers in the newly selected union field are null and therefore valid.

There are two important restrictions that Deputy imposes on the use of tagged unions. First, they must be embedded in structures that contain one or more fields that can be used as tags for the union. Second, one cannot take the address of a union field, although it is possible to take the address of the structure in which it is embedded.

Other C features. Deputy supports a number of other common C features that we do not discuss in detail in this paper. For example, Deputy will use format strings to determine the types of the arguments in printf-style

functions. Also, Deputy allows the program to annotate open arrays (i.e., structures that end with a variable-length array whose length is stored in another field of the structure). Finally, Deputy provides special annotations for functions like memcpy () and memset (), which require additional checks beyond those used for the core Deputy annotations.

3.2 Annotating Programs

The Deputy type system expects to see a pointer bound annotation on every pointer variable in the program in order to insert checks. However, since adding annotations to every pointer type would be difficult and would clutter the code, Deputy provides a simple inference mechanism. For any local variable of pointer type and for any cast to a pointer type, Deputy will insert two new variables that explicitly hold the bounds for this type. The type can then be automatically annotated with bound (b, e), where b and e are the two new variables. Whenever this variable is updated, Deputy inserts code to update b and e to hold the bounds of the righthand side of the assignment. Although this instrumentation inserts many additional assignments and variables, the trivial ones will be eliminated by the Deputy-specific optimizer.

We also use the inference algorithm from CCured [24] to avoid inserting unnecessary local variables. For example, if a local variable is incremented but not decremented, we insert a new variable for the upper bound only. This inference algorithm is also useful for inferring nullterm annotations.

For pointer types other than those found in local variables or casts, Deputy requires the programmer to insert an annotation. Such pointer types include function prototypes, structure fields, and global variables. These annotations are required in order to support separate compilation; since Deputy cannot instrument external function arguments or structures fields the same way it can instrument local variables, the programmer must explicitly supply bounds annotations. Fortunately, most of these annotations appear in header files that are shared among many compilation units, which means that each additional annotation will benefit a large number of modules in the program. Also, Deputy can optionally use a default value for unannotated pointers in the interface (usually safe); however, such annotations are unreliable and should eventually be replaced by an explicit annotation. The inference algorithm assists in this process as well by identifying global variables, functions, and structure fields that require explicit annotations.

3.3 Limitations and Future Work

The most significant safety issue that is ignored by Deputy is the issue of memory deallocation. Deputy is designed to detect safety violations due to bounds violations, but it does not check for dangling pointers, since detecting such violations would require more extensive run-time checking or run-time support than we currently provide. Fortunately, the issue of dangling pointers is largely orthogonal to the problems that Deputy solves; thus, the programmer can choose to use a conservative garbage collector or to simply trust that the program's deallocation behavior is correct.

In addition, there are some cases in which Deputy's type system is insufficient for a given program. First, Deputy's type system is sometimes incapable of expressing an existing dependency; for example, programmers sometimes store the length of an array in a structure that is separate from the array itself. Second, Deputy's type system sometimes fails to understand a type cast where there are significant differences between the pointer's base type before and after the cast. Based on our experience with device drivers (see Section 5) and our previous experience with CCured [24], such casts occur at about 1% of the assignments in a C program. In both of these cases, Deputy provides a trusted cast mechanism that allows the programmer to suppress any Deputy errors or run-time checks for a particular cast or assignment, effectively flagging this location for more detailed programmer review. One of our primary goals in designing Deputy and instrumenting programs is to minimize the number of such casts required; however, in many of these cases, a failure in Deputy's type system indicates a lack of robustness in the code itself. Thus, the process of annotating code can help identify places where the interface between modules can be improved.

Deputy does not provide any special handling for multithreaded code, since correctly synchronized code will still be correctly synchronized after being compiled with Deputy. However, C code that was previously atomic may no longer be atomic due to the addition of run-time checks. Thus, the programmer must ensure that their code makes no unwarranted assumptions about the atomicity of any particular piece of C code.

Finally, Deputy's changes to the source code (e.g., temporary variables and run-time checks) can make source-level debugging more challenging. However, with sufficient engineering effort, code compiled with Deputy could be viewed in the debugger with the same ease as code compiled by GCC. If necessary, code can currently be compiled and debugged with GCC first, using Deputy only for the final stages of development.

In the future, we hope to provide support for many of the programming idioms that currently require trusted casts. For example, the Linux container_of macro, which subtracts from a pointer to a field to get a pointer to the containing object, is a construct we hope to support in a future version. We also plan to improve our handling of void* pointers, particularly for void* pointers that appear in kernel data structures in order to hold private data for a module. And finally, we plan to improve Deputy's optimizer in order to further reduce the number of run-time checks required. Our long-term goal is to allow programmers to migrate drivers (and even the kernel itself) to a type-safe version of C without significantly rewriting their code or affecting performance.

4 Recovery System

This section describes how SafeDrive tracks updates from the driver and recovers from driver failures. We also compare our recovery mechanisms to that of Nooks.

4.1 Update Tracking

The update tracking module maintains a linked list of all updates a driver has made to kernel state that should be undone if the driver fails. For each update, the list stores a compensation operation [36], which is a pointer to a function that can undo the original update, along with any data needed by this compensating action. For example, in Linux device drivers, the compensation function for kmalloc() is kfree(). This list is also indexed by a hash table, which allows compensations to be removed from the list if the driver manually reverses the update (e.g., if an allocated block is freed). SafeDrive provides wrappers for all functions in the kernel API that require update tracking, allowing drivers to use this feature with minimal changes to their source code.

In a few cases, we need to modify the kernel to handle changes to the list of updates that are not explicitly performed by the driver. For example, timers are removed from the list after the corresponding timer function executes.

Updates recorded by the tracking module are divided into two separate pools, one associated with the driver itself (long-term updates) and the other associated with the current CPU and control path (short-term updates). The latter pool holds updates like spinlock-acquires, which have state associated with the local CPU and must be undone atomically (without blocking) on the same CPU.

4.2 Failure Handling

Failures are detected by the run-time checks inserted by the Deputy compiler. When a run-time checks fails, it invokes the SafeDrive recovery system. First, a description of the error is printed for debugging purposes. For problems due to memory safety bugs, this error report pinpoints the actual location where a pointer leaves its designated bounds or is dereferenced inappropriately.

Then SafeDrive goes through a series of steps to clean up the driver module itself, restoring kernel state and optionally restarting the driver, while at the same time allowing other parts of the system to continue running. We maintain two invariants during the recovery process, both vital to the success of recovery:

- Invariant 1: No driver code is executed from the point when a failure is detected until recovery is complete. This invariant is required because the driver is already corrupt; executing driver code could easily trigger more failures or corrupt kernel state.
- Invariant 2: No kernel function is forcefully stopped and returned early. Forceful returns from kernel functions would corrupt kernel state. On the other hand, the driver function that fails is always stopped forcefully, in order to maintain the first invariant.

Returning gracefully from a failed driver. The basic mechanism for "forceful return" from a driver function is a setjmp()/longjump() variant that unwinds the stack and jumps directly to the next instruction after setjmp(). SafeDrive requires the programmer to identify driver entry points, and at these entry points, SafeDrive adds a standard wrapper stub that calls setjmp() to store the context in the current task structure (Linux's kernel thread data structure). When a failure occurs, the failure-handling code will call longjmp () to return to the wrapper, which then returns control to the kernel with a pre-specified return value, often indicating a temporary error or busy condition. The failure-handling code also sets a flag that indicates that the driver is in the "failed" state. This flag is checked at each wrapper stub to ensure that any future calls to the driver will return immediately, thus preserving Invariant 1. This approach allows the kernel to continue normally when the driver fails.

Typically, identifying driver entry points is a simple task; to a first approximation, we can look for all driver functions whose address is taken. Determining the appropriate return value for a failure can be more difficult, since returning an inappropriate value can cause the kernel to hang. Fortunately, the appropriate return value is relatively consistent across each class of drivers in Linux. The Nooks authors provide a more extensive study of possible strategies for selecting proper return values [32].

The recovery process is more complicated in cases where the driver calls back into the kernel, which then calls back into the driver, resulting in a stack that contains interleaved kernel and driver stack frames. If we jump to the initial driver entry point, we skip important kernel code in the interleaved stack frame, violating Invariant 2. However, if we jump to the closest kernel stack frame, we must ensure that Invariant 1 is maintained when we return to the failed driver. To solve this problem, SafeDrive records a context at each re-entry point into the driver. A counter tracks the level of re-entries into the driver, which is incremented whenever the driver calls a kernel function that may call back into the driver. After the driver fails, control jumps directly to re-entry points when it returns from these kernel functions. Essentially, we finish executing any kernel code still on the stack while skipping any driver code still on the stack. This technique is similar to the handling of domain termination in Lightweight RPCs [4], although in our case, both the kernel and the module run in the same protection domain.

Restoring kernel state and restarting driver. During the recovery process, all updates associated with the failed driver are undone in LIFO order by calling the stored compensation functions. These compensations undo all state changes the driver has made to the kernel so far, similar to exception handling in languages like C++ and Java. The main difference between our approach and C++/Java exceptions is that the compensation code does not contain any code from the extension itself, thus preserving Invariant 1. As a result, the extension will not have an opportunity to restore any local invariants; however, because the extension will be completely restarted, we are only concerned with restoring the appropriate kernel invariants. Note that this unwinding task is complicated by the fact that it is executed in parallel with other kernel code and by the fact that the failure could have happened in an inconvenient place, such as an interrupt handler or timer callback. Thus, after CPUlocal compensations such as lock releases are undone in the current context, all other compensations are deferred to be released in a separate kernel thread (in event/*).

After compensations have been performed, the driver's module is unloaded. As mentioned above, we do not call the driver's deinitialization function; however, because we track all state-changing functions provided by the kernel, including any function that registers new devices or other services, calling the driver's deinitialization function should not be necessary. After this process is complete, depending on user settings, the driver can be restarted automatically from a clean slate. The restart process follows the normal module initialization process.

The current SafeDrive update tracking and recovery code is a patch to the Linux kernel changing 1084 lines, tracking in total 21 types of Linux kernel resources for the recovery of four drivers in three classes: two network card drivers, one sound card driver, and one USB device

driver (see Section 5 for details). The resources tracked include 4 types of locks, 4 PCI-related resources, and 4 network-related resources, among others.

4.3 Discussion

The recovery system of SafeDrive resembles that of Nooks [33]. Both track different types of kernel updates and undo them at failure time. However, the fact that all code using SafeDrive runs in the same protection domain makes SafeDrive's recovery system significantly simpler and less intrusive. In fact, the SafeDrive kernel patch is less than one tenth the size of Nooks'. Update-tracking wrappers and compensation functions are simpler mainly because there is no need for code to copy objects in and out of drivers, to manage the life cycles of separate protection domains, or to perform cross-domain calls. In addition, implementing the cross-domain calls efficiently can complicate the system design significantly. For example, Nooks uses complicated page table tricks to enable fast writes of large amount of data from the device driver to the kernel. Nooks also gives each domain its own memory allocation pool, which requires even more changes to the kernel. We believe that simpler recovery code adds significantly to the trustworthiness of the whole mechanism since testing is less effective for such code than for the normal execution path.

The fact that SafeDrive lets drivers modify kernel data structures directly has some ramifications for kernel consistency, which contrasts with Nooks' call-by-valueresult object passing. For each driver call, Nooks first copies all objects the driver may modify, then lets the driver work on the copies, and finally copies the results back. This approach provides atomic updates of kernel objects and thus should provide more consistency. However, we have found there are caveats to this approach. First, it has problems on SMP systems because other processors see the updates to kernel data structures at a different time than they would ordinarily see the updates; the original Nooks paper suggests that SMP is not supported yet [33, end of Sec. 4.1]. Second, many kernel data structure updates are not done through this mechanism but through cross-domain calls to kernel functions. The interaction of these two mechanisms becomes complicated quickly.

Our recovery scheme is orthogonal to the Deputy type system and can also work with other languages, including type-safe languages such as Java. As far as we know, current type-safe languages and runtimes do not provide a general mechanism for recovering buggy extensions. Our technique for gracefully returning from a failed extension should apply here. More broadly, a recovery module based on compensation stacks [36] would be a valuable asset for these systems.

Driver	Description			
e1000	Intel PRO/1000 network card driver			
tg3	Broadcom Tigon3 network card driver			
usb-storage	USB mass-storage driver			
intel8x0	Intel 8x0 builtin sound driver			
emu10k1	Creative Audigy 2 sound driver			
nvidia	NVidia video driver			

Table 1: Drivers used in experiments.

State and session restoration for failed drivers is not yet implemented in our prototype. Thus, the driver will be in an initial empty state after recovery. However, we believe the shadow driver approach proposed by the Nooks project [32] should apply to our system. Its implementation should also be simpler because of the absence of multiple hardware protection domains.

5 Evaluation

In this section, the recovery mechanism is first evaluated in terms of successful recoveries in the face of randomly injected faults. Then we measure two distinct types of overhead of using SafeDrive: (1) the one-time overhead of annotating APIs and adding wrapper functions to a particular extension, and (2) runtime overhead due to additional checks inserted by the Deputy type system and due to update tracking for recovery. We quantify the first one by noting how much of the kernel API and wrappers needed alteration to support a handful of drivers that we chose from different subsystems of the kernel. We quantify the second source of overhead with traditional performance benchmarks.

Our experiments are done with six device drivers in four categories for the Linux 2.6.15.5 kernel, as shown in Table 1. All drivers are annotated with Deputy annotations to detect type-safety errors. However, due to time constraints, we only added update tracking and recovery support to the four drivers with names shown in bold.

5.1 Recovery Rate

Here we evaluate how well SafeDrive is able to handle various faults in drivers. We use *compile-time software-implemented fault injection* to inject random faults into the driver, and then we run the driver with and without SafeDrive. We wrote a compile-time fault injection tool as an optional phase in the Deputy compiler. The tool injects into C code seven categories of faults, shown in Table 2, following two empirical studies on kernel code [10, 31]. We did not use an existing binary-level fault injection tool, such as that used by the Rio file cache [26] or Nooks [34], because all of Deputy's checks are inserted at compilation time, which means that Deputy

Fault Category	Code Transformation			
Loop fault	Make loop count larger by: 1 with 0.5 prob., 2–1024 with 0.44 prob., and 2K–4K with 0.06 prob.			
Scan overrun	Make size parameter to memset-like functions larger as the line above			
Off-by-one	Change < to <=, etc., in boolean expressions			
Flipped condition	Negate conditions in if statements			
Missing assignment	Remove assignments and initialization of local vars			
Corrupt parameter	Replace a pointer parameter with null, or a numeric parameter with a random number			
Missing call	Remove calls to functions and return a random result as in the line above			

Table 2: Categories of faults injected in recovery experiments.

	C	orrect	Incorrect		
SafeDrive	Works	Innocuous Errors	Crashes	Mal- functions	
Off	75	n/a	44	21	
On	113	8	0	19	

Table 3: Results of 140 runs of e1000 with injected faults, with SafeDrive off and on. "Correct" means that the driver behaved as expected, possibly with the help of SafeDrive's recovery subsystem and assuming the bugs are transient.

Detection	Crashes	Malfunc.	Innocuous	Total
Static	10	0	3	13 (24%)
Dynamic	34	2	5	41 (76%)
Total	44	2	8	54

Table 4: Breakdown for the 54 cases in which SafeDrive detected errors.

does not have a chance to catch errors introduced via binary fault injection. Compile-time fault injection allows us to evaluate Deputy's error detection fairly, and it also has the benefit of being able to introduce more realistic programming errors.

The fault-injection experiments were done with e1000 driver. For each experiment, 5 random faults of the same category were inserted into the driver code during the compilation process. A script exercised the driver by loading it, configuring networking, downloading a large (89MB) file, checksumming the file, and finally unloading the driver. Then the same script was run with the unmodified e1000 driver to check whether the system was still functioning. This test was performed with and without SafeDrive recovery on. When SafeDrive recovery was off, Deputy checks were still performed, but they did not trigger any action when failures were detected. Thus the driver should have behaved exactly the same as the original one with faults injected.

Table 3 shows the results of all 140 runs, with 20 runs per fault category. When SafeDrive is disabled, 44 of these runs resulted in crashes and 21 resulted in the driver malfunctioning. When SafeDrive was enabled, SafeDrive successfully prevented all 44 crashes, and it invoked the recovery subsystem on 2 of the 21 non-crash driver malfunctions. In addition, there were 8 runs where

SafeDrive successfully detected apparently innocuous type-safety errors that did not trigger crashes or malfunctions with SafeDrive disabled, but failed Deputy checks when SafeDrive was on.

A closer look at the results reveals that, among the 7 categories of faults injected, all categories except "off-by-one" and "flipped condition" resulted in crashes when SafeDrive was off. The fact that SafeDrive prevented these crashes indicates that these crashes were actually due to type-safety errors, not other serious errors such as incorrect interrupt commands. On the other hand, the malfunctioning runs were due to a variety of reasons, including setting hardware registers to bad values and incorrect return value checking. Type-safety checks alone cannot always detect these errors.

Overall, SafeDrive detected problems in 54 of the runs. Table 4 shows that 24% of the problems were detected statically by the Deputy compiler. These are errors that were *not* detected by GCC, and in fact 10 of them led to crashes. These compile-time errors include passing an incorrect constant size argument to memcpy and dereferencing an uninitialized pointer, among others.

Of the innocuous errors, five were detected dynamically, causing SafeDrive to invoke recovery and successfully restart the driver. Although these errors appear to be innocuous, they are likely latent bugs; thus, invoking recovery seems to be a reasonable response. Of course, SafeDrive has no way to know which errors will be truly benign.

These results show that SafeDrive is effective in detecting and recovering from typical memory and typesafety errors in drivers. In addition, a significant portion of these errors are caught at compile time, before the driver is even run. Finally, it seems unlikely that SafeDrive will always prevent *all* crashes, as it did with the e1000, due to the limits of type checking.

5.2 Annotation Burden

Table 5 shows the number of lines of code we changed in order to use SafeDrive on these drivers. The third column shows all Deputy-related changes, and the next four columns show the number of lines containing each type of Deputy annotation. These numbers do not add up to

Driver Original	Original LOC	Deputy Changes					December Changes
	Original LOC	LOC Modified	Bounds	Nullterm	Tagged Unions	Trusted Code	Recovery Changes
e1000	17011	260	146	15	2	47	270
tg3	13270	359	78	9	0	64	156
usb-storage	13252	136	16	11	0	21	118
intel8x0	2897	124	31	2	0	8	167
emu10k1	11080	441	66	11	0	23	n/a
nvidia	10126	224	42	35	0	27	n/a

Table 5: Number of lines changed in order to enable SafeDrive for each driver. All numbers are lines of code. "LOC Modified" in "Deputy Changes" are number of lines with Deputy annotations and related changes. The next four columns show numbers of lines with each category of annotations. "Recovery Changes" shows the number of lines where trivial wrappers were added for recovery.

the previous number because not every changed line contains a Deputy annotation. These changes are essential to driver safety, and they amount to approximately 1–4% of the total number of lines in a given driver. The last column shows additional recovery-related changes currently needed in the driver code by the SafeDrive prototype. These changes are boilerplate code placed at driver entry points in the driver source files. We intend to generate these wrappers automatically in the future.

The other kind of changes required are changes to the kernel headers. This set of changes includes Deputy annotations for the kernel API as well as wrappers for functions that are tracked by the SafeDrive runtime system. Both types of changes must be written by hand; however, these changes are a one-time cost for drivers of a given class. For the 4 classes of drivers we worked on, a total of 1,866 lines in 112 header files were changed. Casual inspection reveals that about half of the lines are Deputy-related and that the rest are recovery-related. In particular, there are 187 lines with bounds annotations, 260 lines with nullterm, 8 lines with tagged unions, and 140 lines with trusted code annotations.

5.3 Performance

The run-time overhead of SafeDrive is composed of several parts, including run-time checks inserted by the Deputy compiler, update tracking cost, and context saving cost. We measure the performance overhead of SafeDrive by running several benchmarks with native and SafeDrive-enabled drivers.

Table 6 shows results for these benchmarks on a dual Xeon 2.4Ghz. In "TCP Receive", netperf [18] is run on another host and sends TCP streaming data to the testing server (the TCP_STREAM test). The socket buffers are 256KB on both the sending and receiving side, and 32KB messages are sent. "TCP Send" works the other way around, with the testing server running netperf and sending traffic. In "UDP Receive" and "UDP Send", UDP packets of 16 bytes are received/sent (the UDP_STREAM test). CPU utilization is measured

with the sar utility. CPU utilization maxes out at 50%, probably because the driver cannot utilize more than one CPU. The UDP tests show higher overhead probably due to two reasons. First, the packets are much smaller, leading to more Deputy overhead overall. Second, less other kernel code is involved in UDP processing compared to TCP, amplifying SafeDrive's overhead.

We tested the usb-storage driver with a 256MB Sandisk USB2.0 Flash drive on a Thinkpad T43p laptop (2.13Ghz Pentium-M CPU). The "Untar" benchmark simply untars a Linux 2.2.26 source code tar ball, which is already on the drive, to the drive itself. The tar file is 82MB in size. After untarring finishes, the drive is immediately unmounted to flush any data in the page cache. CPU utilization is the average value over the whole period. As can be seen from the results, the whole operation finishes in the same amount of time, though SafeDrive's instrumentation increased CPU usage by 23%.

We also benchmarked two sound card drivers: the intel8x0 sound driver for the built-in sound chip in a Thinkpad T41 (1600Mhz Pentium-M CPU), and the emu10k1 sound driver for a Creative Audigy 2 card on a Pentium II 450Mhz PC. Both benchmarks used the oprofile facility to capture how much time (in percentage of total CPU time) was spent in the kernel on behalf of the sound driver while a 30-minute 44.1Khz wave file was playing. aplay and the standard alsa sound library were used for playback. Throughput is irrelevant here because the sample rate is fixed.

Finally, we tested the open-source portion of the driver distributed by NVidia for their video cards. These 10,296 lines of open-source code are the interface between the kernel and a larger, proprietary graphics module which we did not process because the source code is not available. We tested the nvidia driver on a Pentium 4 2.4 GHz machine with a GeForce4 Ti 4200 graphics card. Table 6 shows the CPU usage of this driver while setting up and tearing down an X Window session, as measured by oprofile. The instrumentation did not have a measurable effect on the performance of the x11perf graphics benchmarking tool, which is limited by hard-

Benchmark	Driver	Native Throughput	SafeDrive Throughput	Native CPU %	SafeDrive CPU %	
TCP Receive	e1000	936Mb/s	936Mb/s	47.2	49.1 (+4%)	
UDP Receive	e1000	20.9Mb/s	17.4Mb/s (-17%)	50.0	50.0	
TCP Send	e1000	936Mb/s	936Mb/s	20.1	22.5 (+12%)	
UDP Send	e1000	33.7Mb/s	30.0Mb/s (-11%)	45.5	50.0 (+9%)	
TCP Receive	tg3	917Mb/s	905Mb/s (-1.3%)	25.4	27.4 (+8%)	
TCP Send	tg3	913Mb/s	903Mb/s (-1.1%)	18.0	20.4 (+13%)	
Untar	usb-storage	1.64MB/s	1.64MB/s	5.5	6.8 (+23%)	
Aplay	emu10k1	n/a	n/a	9.10	9.64 (+6%)	
Aplay	intel8x0	n/a	n/a	3.79	4.33 (+14%)	
Xinit	nvidia	n/a	n/a	12.13	12.59 (+4%)	

Table 6: Benchmarks measuring SafeDrive overhead. Utilization numbers are kernel CPU utilization.

ware performance rather than by the driver.

The above results show that SafeDrive has relatively low performance overhead, even for data-intensive drivers. To compare with Nooks, consider the e1000 TCP send/receive tests. These tests are similar to experiments discussed in the Nooks journal paper [34], where the authors reported relative receive and send CPU overheads of 111% and 46% respectively, both with 3% degradation of throughput. This overhead is nearly an order of magnitude higher than the overhead of SafeDrive (rows 1 and 3 in Table 6), suggesting that the overhead of cross-domain calls far outweighs the overhead of Deputy's run-time checks for these benchmarks.

6 Related Work

6.1 Enforcing Isolation with Hardware

Several projects have used hardware to isolate device drivers from the rest of the system and to allow recovery in the case of failure.

The Nooks project [32, 33] isolates device drivers from the main kernel by placing them in separate hardware protection domains called "nooks". These protection domains share the same address space but have different permission settings for pages. A driver is permitted to read all kernel data but only allowed to write to certain pages. Cross-domain calls replace function calls between the driver and the kernel, although the semantics of the calls remain mostly similar.

Virtual Machine Monitors (VMMs) such as L4 [21] and Xen [3, 13] isolate a driver using hardware protection. However, rather than placing a protection barrier between a driver and the kernel, the VMM runs each driver with its own kernel inside a separate virtual machine. This approach allows device drivers to be used unchanged, and it allows one to use device drivers that depend on different operating systems. Communication

between virtual machines is performed using a specialpurpose high-performance batched channel interface.

From a pure driver isolation standpoint, SafeDrive is less secure than a VMM since it is possible for a driver to manipulate the kernel via the API calls, privileged instructions, or simply tying up CPU resources by looping. In the VMM case, however, a buggy driver can only corrupt the driver's local, untrusted kernel, which is isolated from the trusted kernel behind a message-passing interface. The SLAM project [2] looks at API usage validation, and in theory could close this gap when combined with SafeDrive.

However, this caveat does not mean that SafeDrive is able to catch fewer errors than a VMM. In fact, one advantage of SafeDrive is that its finer-grained checks are able to detect errors within the driver and not just outside it. Although hard to measure, this fine-grained error detection reduces the likelihood of data corruption and is very important in cases involving persistent data. For example, a misbehaving disk driver in any of Xen, Nooks, or SFI can corrupt data on disk before the fault is detected. This exact behavior occurred with Nooks during their fault injection experiments [33].

Another advantage of SafeDrive relative to these systems is performance. The additional domain crossings required by the hardware approaches impose additional costs. In all three hardware-based systems, one can generally expect the CPU overhead for data intensive device drivers to be between 40% to 200% [21, 22, 32, 34]. This result contrasts with a typical CPU overhead of less than 20% for SafeDrive, which incurs no additional cost for calls into or out of a driver. Of course, it is not guaranteed that SafeDrive will always outperform hardware approaches: if crossings are rare and checks are frequent then a hardware solution is likely to outperform SafeDrive; however, our experiments suggest that SafeDrive performs better in practice and that SafeDrive's performance is likely to improve further as its optimizer improves.

6.2 Enforcing Isolation with Binary Instrumentation

Software-enforced Fault Isolation (SFI) [11, 29, 35] instruments extension binaries to ensure that no memory operation can write outside of an extension's designated memory region. The instrumentation can take one of several forms, depending on the desired tradeoff between isolation and performance.

If reads are protected as well as writes (as they are in SafeDrive), then typical performance overhead varies between 17% and 144%, depending on the SFI implementation and the benchmarks being used. Although it is hard to make direct performance comparisons against results obtained from different test programs, we expect SafeDrive to exceed the performance of SFI, since SafeDrive only needs to check memory accesses for which the type checker is unable to verify correctness.

As with the hardware-based approaches, SFI only prevents an extension from corrupting the system, and it does not attempt to prevent a driver from corrupting itself or the device. Furthermore, these approaches require a very clean kernel-driver interface in order to ensure that all data passed between the kernel and the driver can be checked at run time [11].

6.3 Enforcing Isolation with a Language

A number of research projects have attempted to enforce memory safety in C programs at source level instead of at binary level. CCured [24], a predecessor to Deputy, used a whole-program analysis to classify pointers according to their use, and then it altered data structures and code to provide low-cost run-time memory safety checks. Unfortunately, CCured made significant changes to the program's data structures, making it difficult to apply CCured to one module at a time. Another sourcelevel tool is the Cyclone [19] language, which is a safe alternative to the C language that has been used to write safe kernel-level code [1]. However, like CCured, Cyclone requires a large amount of manual intervention to port existing drivers when compared to Deputy. Neither CCured nor Cyclone support a priori data layouts, which are a prerequisite for extensions with predefined APIs and data structures. Finally, Yong and Horwitz [38] use static analysis to insert efficient buffer overflow checks; however, they do not address memory safety in general, and they provide relatively coarse-grained checks.

The major advantage of the Deputy type system over these other source-level approaches is that Deputy allows the programmer to describe pointer bounds in terms of other variables or fields in the program, and thus Deputy can leave data layout and APIs unchanged. A related project at Microsoft uses the SAL annotation language and the ESPX modular annotation checker in order to find buffer overflows [14]. Although SAL can describe relationships between variables, it cannot describe relationships between structure fields, and it does not support tagged union types. Also, ESPX is a static analysis, which means that problematic code is simply left unverified; in contrast, Deputy inserts run-time checks where static analysis is insufficient.

There are also a number of related projects that allow types to refer to program data, including the Xanadu language [37] and Hickey's very dependent function types [16]. However, these projects have a number of restrictions on mutable data, which Deputy addresses using run-time checks. Also, Harren and Necula [15] developed a similar framework for assembly language in which dependencies can occur between registers or between structure fields.

Type qualifiers represent another area of related work. CQual [12, 20] allows programmers to add custom type qualifiers such as const or user/kernel, using an inference algorithm to propagate these qualifiers throughout the program. Semantic type qualifiers [8] builds on this work by allowing qualifiers to be proved sound in isolation. Compared to this work, Deputy's annotations are more expressive (since annotations can refer to other program data) and correspondingly more difficult to infer. Although Deputy provides a number of features for inferring or guessing annotations, it relies more heavily on programmer-supplied annotations than these other type qualifier tools. Finally, Privtrans [6] allows the programmer to specify privileged operations and automatically separates a program into a privileged process and non-privileged process, improving security.

There are operating systems built mainly with type-safe languages, such as Singularity [17], JavaOS [23], and the Lisp Machine OS [30]. These operating systems naturally have few memory-safety problems, and as processor speed increases, the performance penalty of these languages become less of a concern. While this approach will be important in building future operating systems, we believe that current commodity operating systems such as Windows and Linux, which are written mainly in C and C++, will be in wide use for many years. SafeDrive will be useful both in improving the reliability of these existing operating systems and in providing a transition to future type-safe operating systems.

6.4 Fault Tolerance for Applications

In addition to the issue of how a device driver should be isolated from the rest of the system, there is also the largely orthogonal issue of how the system should recover when a driver failure is detected. Both the original Nooks paper [33] and SafeDrive provide isolation, release of resources, and restart of the driver.

A later Nooks paper [32] showed how to use *shadow drivers* to restore the session state of applications that were using the driver. We expect the shadow driver technique to work without significant modification with SafeDrive, and thus we do not address this issue further.

Vino [28], which isolates drivers using SFI, executes each driver request inside a transaction that can be aborted and retried on failure. Another related technique is Microreboot [7], which is used to build rebootable components in large enterprise systems. Session restoration is achieved by programming the components against a separate session state storage that persists over component restarts.

7 Conclusion

We have presented SafeDrive, a system that uses language-based techniques to detect type safety errors and to recover from such errors in device drivers written in C. The checking in SafeDrive is fine-grained, which is critical because it not only protects the kernel from misbehaving drivers, but also helps prevent the driver from corrupting persistent data or kernel state. SafeDrive requires few changes to the kernel or drivers, and experiments show that SafeDrive incurs low overhead (normally less than 20%) and successfully prevents all 44 crashes due to randomly injected errors for one driver. Overall, we hope that this work shows that we can achieve the safety of high-level, type-safe languages without abandoning existing C code.

Acknowledgments

We thank the anonymous reviewers and our shepherd Dawn Song for their useful suggestions and comments on the paper. We also thank David Gay for insightful discussion.

Notes

¹This requirement is slightly different from the official strlcpy specification, since dst is required to be null-terminated on entry as well as on exit. However, in practice, establishing this invariant on entry should not require much, if any, additional effort on the part of client code.

References

- ANAGNOSTAKIS, K., GREENWALD, M., IOANNIDIS, S., AND MILTCHEV, S. Open packet monitoring on FLAME: Safety, performance and applications. In *Proceedings of the 4rd Inter*national Working Conference on Active Networks (IWAN 2002) (2002).
- [2] BALL, T., MAJUMDAR, R., MILLSTEIN, T. D., AND RAJA-MANI, S. K. Automatic predicate abstraction of C programs.

- In SIGPLAN Conference on Programming Language Design and Implementation (2001).
- [3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings* of the 19th ACM Symposium on Operating Systems Principles (2003).
- [4] BERSHAD, B., ANDERSON, T., LAZOWSKA, E., AND LEVY, H. Lightweight remote procedure call. In Proceedings of the 12th ACM Symposium on Operating Systems Principles (1989).
- [5] Bos, H., AND SAMWEL, B. Safe kernel programming in the OKE. In Open Architectures and Network Programming Proceedings (2002).
- [6] BRUMLEY, D., AND SONG, D. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the* 13th USENIX Security Symposium (2004).
- [7] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot — a technique for cheap recovery. In Symposium on Operating System Design and Implementation (2004).
- [8] CHIN, B., MARKSTRUM, S., AND MILLSTEIN, T. Semantic type qualifiers. In Proceedings of the ACM Conference on Programming Language Design and Implementation (2005).
- [9] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating system errors. In *Proceedings* of the 18th ACM Symposium on Operating Systems Principles (2001).
- [10] CHRISTMANSSON, J., AND CHILLAREGE, R. Generation of an error set that emulates software faults — based on field data. In Proceedings of the 26th IEEE International Symposium on Fault Tolerant Computing (1996).
- [11] ERLINGSSON, Ú., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software guards for system address spaces. In Symposium on Operating System Design and Implementation (2006).
- [12] FOSTER, J. S., FAHNDRICH, M., AND AIKEN, A. A theory of type qualifiers. In Proceedings of the ACM Conference on Programming Language Design and Implementation (1999).
- [13] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND MARK WILLIAMSON. Safe hardware access with the Xen virtual machine monitor. In Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS 2004) (2004).
- [14] HACKETT, B., DAS, M., WANG, D., AND YANG, Z. Modular checking for buffer overflows in the large. Technical Report MSR-TR-2005-139, Microsoft Research, 2005.
- [15] HARREN, M., AND NECULA, G. C. Using dependent types to certify the safety of assembly code. In *Proceedings of the 12th* international Static Analysis Symposium (SAS) (2005).
- [16] HICKEY, J. Formal objects in type theory using very dependent types. In Proceedings of the 3rd International Workshop on Foundations of Object-Oriented Languages (1996).
- [17] HUNT, G., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FAHNDRICH, M., HAWBLITZEL, C., HODSON, O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., WOBBER, T., AND ZILL, B. D. An overview of the Singularity project. Tech. rep., Microsoft Research, 2005.
- [18] INFORMATION NETWORKS DIVISION, H.-P. C. Netperf: A network performance benchmark, http://www.netperf.org.
- [19] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CH-ENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In USENIX Annual Technical Conference (2002).

- [20] JOHNSON, R., AND WAGNER, D. Finding user/kernel pointer bugs with type inference. In USENIX Security Symposium (2004).
- [21] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GOTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In Symposium on Operating System Design and Implementation (2004).
- [22] MENON, A., SANTOS, J. R., TURNER, Y., G. (JOHN) JANAKI-RAMAN, AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In Proceeding of the 1st ACM/USENIX Conference on Virtual Execution Environments (VEE 2005) (2005).
- [23] MITCHELL, J. G. JavaOS: Back to the future (abstract). In Symposium on Operating System Design and Implementation (1996).
- [24] NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. CCured: Type-safe retrofitting of legacy software. ACM Transactions on Programming Languages and Systems 27, 3 (2005).
- [25] NECULA, G. C., MCPEAK, S., AND WEIMER, W. CIL: Intermediate language and tools for the analysis of C programs. In International Conference on Compiler Construction (2002).
- [26] NG, W. T., AND CHEN, P. M. The systematic improvement of fault tolerance in the Rio file cache. In Symposium on Fault-Tolerant Computing (1999).
- [27] PATEL, P., WHITAKER, A., WETHERALL, D., LEPREAU, J., AND STACK, T. Upgrading transport protocols using untrusted mobile code. In SOSP (2003).
- [28] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In Symposium on Operating System Design and Implementation (1996).
- [29] SMALL, C., AND SELTZER, M. MISFIT: A tool for constructing safe extensible C++ systems. In Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies (COOT 1997) (1997).
- [30] STALLMAN, R., WEINREB, D., AND MOON, D. The Lisp Machine Manual. Massachusetts Institute of Technology, 1981.
- [31] SULLIVAN, M., AND CHILLAREGE, R. Software defects and their impact on system availability — a study of field failures in operating systems. In *Proceedings of the 21st International* Symposium on Fault-Tolerant Computing (1991).
- [32] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering device drivers. In Symposium on Operating System Design and Implementation (2004).
- [33] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *Proceedings* of the 19th ACM Symposium on Operating Systems Principles (2003).
- [34] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. ACM Transactions Computer Systems 23, 1 (2005).
- [35] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. ACM SIGOPS Operating Systems Review 27, 5 (1993).
- [36] WEIMER, W., AND NECULA, G. C. Finding and preventing runtime error handling mistakes. In *Proceedings of the ACM Con*ference on Object-Oriented Programming, Systems, Languages, and Applications (2004).
- [37] X1, H. Imperative programming with dependent types. In Proceedings of 15th IEEE Symposium on Logic in Computer Science (2000).
- [38] YONG, S. H., AND HORWITZ, S. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the* 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering (2003).

BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML

Charles Reis* John Dunagan† Helen J. Wang† Opher Dubrovsky† Saher Esmeir‡

Abstract

Vulnerability-driven filtering of network data can offer a fast and easy-to-deploy alternative or intermediary to software patching, as exemplified in Shield [43]. In this paper, we take Shield's vision to a new domain, inspecting and cleansing not just static content, but also dynamic content. The dynamic content we target is the dynamic HTML in web pages, which have become a popular vector for attacks. The key challenge in filtering dynamic HTML is that it is undecidable to statically determine whether an embedded script will exploit the browser at run-time. We avoid this undecidability problem by rewriting web pages and any embedded scripts into safe equivalents, inserting checks so that the filtering is done at run-time. The rewritten pages contain logic for recursively applying run-time checks to dynamically generated or modified web content, based on known vulnerabilities. We have built and evaluated BrowserShield, a system that performs this dynamic instrumentation of embedded scripts, and that admits policies for customized run-time actions like vulnerabilitydriven filtering.

1 Introduction

Web browsers have become an important interface between users and many electronic services such as information access, personal communications, office tasks, and e-commerce. The importance of web browsers is accompanied by rich functionality and extensibility, which arguably have also contributed to their popularity as a vector of attack. During the year 2005, 8 out of 29 critical Microsoft security bulletins, corresponding to 19 vulnerabilities, are due to flaws in Internet Explorer (IE) or its extensions such as ActiveX controls [1]. There were also 6 security bulletins for Firefox [14], corresponding to 59 vulnerabilities over the same period of time.

To date, the primary way to defend browser vulnerabilities is through software patching. However, studies have shown that the deployment of software patches is ing motivated us to explore its potential for exploit removal in web pages. The Shield approach is able to filter static HTML pages by treating HTML as another protocol layer over HTTP. However, the challenge lies in dynamic HTML, where pages can be dynamically gener-

ated or modified through scripts embedded in the page — attackers could easily evade Shield filters by using scripts to generate malicious web content at run-time, possibly with additional obfuscation. Determining whether a

These desirable features of vulnerability-driven filter-

often delayed after the patches become available. Services such as Windows Update download patches automatically, but typically delay enactment if the patch requires a reboot or application restart. This delay helps both home and corporate users to save work and schedule downtime. An additional delay in the corporate setting is that patches are typically tested prior to deployment, to avoid the potentially high costs for recovering from a faulty patch [5].

As a result, there is a dangerous time window between patch release and patch application during which attackers often reverse-engineer patches to gain vulnerability knowledge and then launch attacks. One study showed that a large majority of existing attacks target known vulnerabilities [4].

For vulnerabilities that are exploitable through application level protocols (e.g., HTTP, RPC), previous work, Shield [43], addresses the patch deployment problem by filtering malicious traffic according to vulnerability signatures at a firewall above the transport layer. The vulnerability signatures consist of a vulnerability state machine that characterizes all possible message sequences that may lead to attacks, along with the message formats that can trigger the exploitation of the application (e.g., an overly long field of a message that triggers a buffer overrun). The key characteristic of this approach is that it cleanses the network data without modifying the code of the vulnerable application. This data-driven approach makes signature deployment (and removal if needed) easier than it is for patches. Vulnerability signature deployment can be automatic rather than user-driven and use the same deployment model as anti-virus signatures.

^{*}University of Washington CS Dept., creis@cs.washington.edu

[†]Microsoft, {jdunagan, helenw, opherd}@microsoft.com

[‡]Technion CS Dept., esaher@cs.technion.ac.il

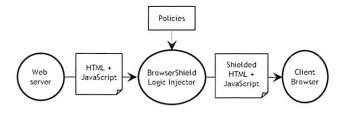


Figure 1: The BrowserShield System

script will eventually exploit a vulnerability is undecidable. Our approach to cleansing dynamic content is to rewrite HTML pages and any embedded scripts into safe equivalents before they are rendered by the browser. The safe equivalent pages contain logic for recursively applying run-time checks to dynamically generated or modified web content, based on known vulnerabilities. To this end, we have designed BrowserShield, a system that performs dynamic instrumentation of embedded scripts and that admits policies for changing web page behavior. A vulnerability signature is one such policy, which sanitizes web pages according to a known vulnerability. Figure 1 gives an overview of the BrowserShield system, showing how it transforms HTML and JavaScript using a set of policies. Our system focuses on JavaScript because it is the predominant scripting language used on the web; a full fledged system would require additionally rewriting or disabling VBScript and any other script languages used by web browsers that BrowserShield protects.

Our general approach of code rewriting for interposition has been used in other contexts. Code rewriting has been used to isolate faults of software extensions [41]. Java bytecode rewriting has been used to enable security polices [10, 37], such as stack inspection policies for access control. However, rewriting script code for web browsers poses additional challenges: JavaScript is a prototype-based language, and the combination of this with JavaScript's scoping rules, implicit garbage collection and pervasive reflection required a number of techniques not needed by previous rewriting work in other contexts.

We have designed BrowserShield to adhere to well established principles for protection systems: complete interposition of the underlying resource (i.e., the HTML document tree), tamper-proofness and transparency [3, 10, 33]. In addition, BrowserShield is a general framework that supports applications other than vulnerability-driven filtering. For example, we have authored policies that add UI invariants to prevent certain phishing attempts.

Because BrowserShield protects web browsers by transforming their inputs, not the browser itself, the BrowserShield logic injector can be deployed at client or edge firewalls, browser extensions, or web publishers that republish third-party content such as ads.

We have implemented a prototype of the Browser-Shield system, in which the rewriting logic is injected into a web page at an enterprise firewall and executed by the browser at rendering time. Our prototype can transparently render many familiar websites that contain JavaScript (e.g., www.google.com, www.cs.washington.edu, www.mit.edu). We also successfully translated and ran a large intranet portal application (Microsoft SharePoint) that uses 549 KB of JavaScript libraries.

We chose the firewall deployment scenario because it offers the greatest manageability benefit, as Browser-Shield updates can be centralized at the firewall, immediately protecting all client machines in the organization without any BrowserShield-related installation at either clients or web servers. The main disadvantage of this deployment scenario is that firewalls have no visibility into end-to-end encrypted traffic. Nevertheless, commercial products [35] already exist that force traffic crossing the organization boundary to use the firewall (instead of a client within the organization) as the encryption endpoint, trading client privacy for aggregate organization security. Also, the browser extension and web publisher deployment scenarios transparently handle encrypted traffic.

Our evaluation focuses on the effectiveness of the BrowserShield design and the performance of our implementation. Our analysis of recent IE vulnerabilities shows that BrowserShield significantly advances the state-of-the-art; existing firewall and anti-virus techniques alone can only provide patch-equivalent protection for 1 of the 8 IE patches from 2005, but combining these two with BrowserShield is sufficient to cover all 8. We evaluated BrowserShield's performance on real-world pages containing over 125 KB of JavaScript. Our evaluation shows a 22% increase in firewall CPU utilization, and client rendering latencies that are comparable to the original page latencies for most pages.

The rest of the paper is organized as follows: In Section 2 we describe a typical browser vulnerability that we would like to filter. We discuss the design of Browser-Shield in Section 3, and give BrowserShield's JavaScript rewriting approach in detail in Section 4. We describe our implementation in Section 5. In Section 6 we give our evaluation of BrowserShield. We discuss related work in Section 7, and conclude in Section 8.

2 A Motivating Example

As a motivating example of vulnerability-driven filtering, we consider *MS04-040*: the HTML Elements Vulnerability [28] of IE from December, 2004. In this vulnerability, IE had a vulnerable buffer that was overrun if

```
function (tag) {
  var len = 255; // not the actual limit

  // Look for long attribute values
  if ((contains("name", tag.attrs) &&
       tag.attrs["name"].length > len) &&
       (contains("src", tag.attrs) &&
       tag.attrs["src"].length > len)) {
       // Remove all attributes to be safe
       tag.attrs = [];
       // Return false to indicate exploit
       return false;
   }
   // Return true to indicate safe tag
   return true;
}
```

Figure 2: JavaScript code snippet to identify exploits of the MS04-040 vulnerability

both the name and the src attributes were too long in an iframe, frame, or embed HTML element.

Figure 2 shows a corresponding snippet of JavaScript code that can be used to identify and to remove exploits of this vulnerability. As input, the function takes an object representing an HTML tag, including an associative array of its attributes. When invoked on an <iframe>, <frame> or <embed> tag, the function determines whether the relevant attributes exceed the size of the vulnerable buffer.

The goal of BrowserShield is to take this vulnerability-specific filtering function as a policy and apply it to all occurrences of the vulnerable tags whether they are in static HTML pages or dynamically generated by scripts. The framework could react in many ways to detected exploits; our current system simply stops page rendering and notifies the user. Vulnerability driven filtering, used as a patch alternative or intermediary, should prevent all exploits of the vulnerability (i.e., zero false negatives), and should not disrupt any exploit-free pages (i.e., zero false positives). We design BrowserShield to meet these requirements.

3 Overview

The BrowserShield system consists of a JavaScript library that translates web pages into safe equivalents and a logic injector (such as a firewall) that modifies web pages to use this library.

BrowserShield uses two separate translations along with policies that are enforced at run-time. The first translation, T_{HTML} , translates the HTML: It tokenizes an HTML page, modifies the page according to its policies (such as the one depicted in Figure 2) and wraps the script elements so that the second translation,

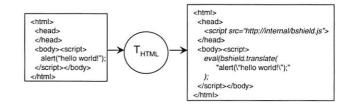


Figure 3: T_{HTML} Translation

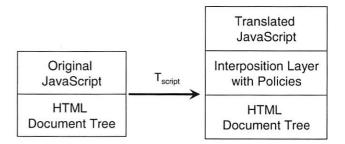


Figure 4: T_{script} Translation

 T_{script} , will be applied at run-time during page rendering at the browser. T_{HTML} is depicted in Figure 3 using bshield.translate(...) to invoke T_{script} . T_{script} , as depicted in Figure 4, parses and rewrites JavaScript to access the HTML document tree through an interposition layer. This layer regulates all accesses and manipulations of the underlying document tree, recursively applies T_{HTML} to any dynamically generated HTML, and recursively applies T_{script} to any dynamically generated script code. Additionally, the interposition layer enforces policies, such as filtering exploits of known vulnerabilities.

Since users can choose to disable scripting in their web browsers, we must ensure BrowserShield protects such users even without the JavaScript library. We transparently handle such clients by applying T_{HTML} at the logic injector, independent of the user's browser. Any modifications due to T_{script} are still in place, but disabling scripts has made them irrelevant, along with the original script code.

Browser extensions, such as ActiveX controls, can also manipulate the document tree. The security model for such extensions is that they have the same privileges as the browser, and thus we focus on interposing between script and the extensions, not between the extensions and the document tree. This allows BrowserShield to prevent malicious script from exploiting known vulnerabilities in trusted browser extensions.

We have designed BrowserShield to adhere to well established principles for protection systems [3, 10, 33]:

- Complete interposition: All script access to the HTML document tree must be mediated by the BrowserShield framework.
- Tamper-proof: Web pages must not be able to modify or tamper with the BrowserShield framework in unintended ways.
- Transparency: Apart from timing considerations and reasonable increases in resource usage, web pages should not be able to detect any changes in behavior due to the BrowserShield framework. The sole exception is for policy enforcement (e.g., the behavior of a page containing an exploit is visibly modified).
- Flexible policies: We desire the BrowserShield framework to have a good separation between mechanism and policy, to make the system flexible for many applications.

4 Design

We now give a detailed discussion of the BrowserShield script library. While much previous work uses code rewriting for interposition [10, 11, 12, 41], our approach is heavily influenced by the fact that our code lives in the same name space as the code it is managing, and also several subtleties of JavaScript. First, JavaScript is a prototype-based language [39], not a class-based language like Java. In prototype-based languages, objects are created using other objects as prototypes, and can then be modified to have a different set of member variables and methods. A consequence of this is that JavaScript has no static typing: different data types can be assigned to the same variable, even for references to functions and object methods. Second, scoping issues must be dealt with carefully, as assigning a method to a new object causes any use of the this keyword in the method to bind to the new object. Thus, any interposition mechanisms must ensure that this is always evaluated in the intended context. Third, JavaScript uses a garbage collector that is not exposed to the language. Fourth, the language has pervasive reflection features that let a script explore its own code and object properties.

As a result of these subtleties, BrowserShield must use a series of interposition mechanisms: *method wrap- pers*, *new invocation syntax*, and *name resolution management*. We justify and describe these mechanisms in the following subsections, organized by our goals for the framework.

4.1 Complete Interposition

To provide complete interposition, BrowserShield must mediate all possible accesses and manipulations allowed by the Document Object Model (DOM) over the HTML document trees (including script elements). In this subsection, we detail how we achieve this using script rewriting to interpose on function calls, object method calls, object property accesses, object creation, and control constructs. We summarize our rewriting rules in Table 1.

Function and Object Method Calls There are two ways to rewrite function or method calls for interposition: callee rewriting or caller rewriting.

In callee rewriting, the original function or method definition is first saved under a different name, and then the original function or method is redefined to allow interception before calling the saved original. We call the redefined function the *wrapper*. The benefit of callee rewriting is that the rewritten code is localized — only functions or methods of interest are modified, but not their invocations throughout the code. However, callee rewriting does not work for cases where functions or methods cannot be redefined.

In caller rewriting, the invocation is rewritten to an interposition function without changing the original function's definition. The interposition function looks up the appropriate interposition logic based on the identity of the target function or method. Although caller rewriting causes more pervasive code changes, it can interpose on those functions or methods that cannot be overwritten.

In BrowserShield, we have to use a hybrid of both approaches to accommodate the previously mentioned JavaScript subtleties.

JavaScript contains some native functions that cannot be redefined (e.g., alert), which necessitates caller rewriting. The first row of Table 1 shows how BrowserShield indirectly invokes a function with its list of parameter values by passing it to bshield.invokeFunc(func, paramList), where bshield is a global object that we introduce to contain BrowserShield library code.

However, using caller rewriting alone for interposing on method calls requires maintaining references to state otherwise eligible for garbage collection. Caller rewriting requires maintaining a map from functions and methods of interest to their associated interposition logic. Maintaining this map as a global table would require maintaining a reference to methods of interest on every object ever created, since each object may correspond to a distinct prototype requiring distinct interposition logic. These global table references would prevent reclamation of objects otherwise eligible for garbage collection, possibly causing pages that render normally without BrowserShield to require unbounded memory. To avoid this, BrowserShield maintains the necessary interposition logic on each method, allowing unused state to be reclaimed.

It might seem tempting to maintain this interposition logic as a property on the object. Unfortunately,

Construct	Original Code	Rewritten Code	
Function Calls	foo(x);	bshield.invokeFunc(foo, x);	
Method Calls	document.write(s);	bshield.invokeMeth(document, "write", s);	
Object Properties	obj.x = obj.y;	bshield.propWrite(obj, "x",	
		<pre>bshield.propRead(obj, "y"));</pre>	
Object Creation	<pre>var obj = new MyClass(x);</pre>	<pre>var obj = bshield.createObj(</pre>	
		"MyClass", [x]);	
with Construct	with (obj) { x = 3; }	(bshield.undefined(obj.x) ? x = 3 :	
	// x refers to obj.x	<pre>bshield.propWrite(obj, "x", 3));</pre>	
Variable Names	bshield = x;	bshield_ = x;	
in Construct	for (i in obj) {}	for (i in obj) {	
	A 100 100 100 100 100 100 100 100 100 10	if (i=="bshield") continue;	
		}	

Table 1: Sample Code for BrowserShield Rewrite Rules

aliases to the interposed method can be created, and these aliases provide no reference to the object containing the interposition logic. For example, after "f = document.write", any interposition logic associated with document.write is not associated with f; finding the logic would require a global scan of JavaScript objects. Therefore, we use callee rewriting to install a wrapper for the methods of interest, such as those that insert new HTML. These wrappers are installed by replacing the original method with the wrapper and saving the original method as a property on the wrapper (which is itself an object). Because we interpose on object property accesses, object creation, and method invocations, we can install wrappers when an object is first created or used.

Thus far we have justified caller rewriting for functions and callee rewriting for methods. Because JavaScript allows functions to be aliased as methods on objects (e.g., "obj.m = eval"), we also must perform caller rewriting for method calls. The rewritten method invocations can then check for potential aliased functions.

JavaScript scoping introduces additional complexity in method interposition. The original method cannot be simply called from the method wrapper, because saving the original method as a property of the wrapper causes the keyword this in the original method to refer to the wrapper rather than the intended object. To avoid this problem, we use a *swapping* technique: The wrapper temporarily restores the original method during the wrapper execution, and then reinstalls the wrapper for the object method before the wrapper returns.

During swapping, the first step is to restore the original method. One challenge here is that the method name may not be the same as when the method wrapper was installed, because methods can be reassigned. We solve this problem again with

caller rewriting through the rewritten method invocation syntax invokeMeth(obj, methName, paramList), passing the name of the method to the method wrapper.

The swapping process requires an additional check to handle recursive methods, since otherwise a recursive call would directly invoke the original method rather than the swapped out method wrapper, bypassing any interposition logic on nested calls. To this end, the invokeMeth method checks to see if a wrapper is already swapped out. If so, invokeMeth invokes the wrapper again, ignoring any swapping logic until the original recursive call completes. Because JavaScript is single threaded, we have not needed to handle concurrency during this process.

Object Properties The HTML document tree can be accessed and modified through JavaScript object property reads and writes. For example, the HTML in a page can be modified by assigning values to document.body.innerHTML, and a script element's code can be modified by changing its text property. To interpose on such actions, BrowserShield replaces any attempts to read or write object properties with calls to the global bshield object's propRead(obj, propName) and propWrite(obj, propName, val) methods, as shown in Table 1. We use an object's identity at run-time to check whether an assignment will create new HTML or script code. If so, propWrite applies either T_{HTML} or T_{script} to the value as needed. These identity checks can be done by calling JavaScript library functions that reveal whether the object is part of the HTML document tree. We ensure that BrowserShield uses the authentic library functions, and not malicious replacements, by creating private aliases of the functions before the script begins to run.

This interposition on property accesses is required for installing wrappers when an object is first accessed. Additionally, while wrappers are swapped out during method execution, propRead must ensure that any attempts to access the original method are redirected to the swapped-out wrapper.

Object Creation To ensure that method wrappers are initialized in the case of new object creation, Browser-Shield must also rewrite the instantiation of new objects to use the createObj (className, paramList) method. The createObj method is also responsible for interposing on the JavaScript Function constructor, which can create new executable functions from its parameters as follows:

```
f = new Function("x", "return x+1;");
```

In this case, createObj applies T_{script} to the code argument before instantiating the function.

Control Constructs For control constructs (e.g., ifthen blocks, loops, etc.), the bodies of the constructs are translated by T_{script} . The bodies of traditional function constructors (e.g., function foo() $\{\ldots\}$) are translated by T_{script} as well.

JavaScript's with construct presents a special case, as it has the ability to modify scope. As shown in Table 1, free variables within a with block are assumed to refer to properties on the designated object, unless such properties are undefined. This construct is purely "syntactic sugar" for JavaScript, and thus we handle this case with a syntactic transformation.

4.2 Tamper-Proof

Preventing scripts from tampering with BrowserShield is challenging because BrowserShield logic lives in the same name space as the code it is managing. To address this, we use *name resolution management* to ensure that all BrowserShield logic is inaccessible.

Variable Names In the common case, variable names in a script can remain unchanged. However, we make the bshield name inaccessible to scripts to prevent tampering with the global BrowserShield data structure.

To do this, we rename any variable references to bshield by appending an underscore to the end of the name. We also append an underscore to any name that matches the bshield (_*) regular expression (i.e., that begins with bshield and is optionally followed by any number of underscores). Note that JavaScript places no limit on variable name length.

Reflection Reflection in JavaScript allows script code to explore the properties of objects as well as its own code, using two pervasive language features: the syntax for accessing object properties (such as myScript.text or myScript[i]), and the JavaScript in construct.

In the first case, BrowserShield must hide some object properties, because it maintains per-object interposition state (details given in Section 4.3) on some objects. Such state is stored on a bshield property of the object, which we hide using property access interposition. Specifically, if a call to propRead or propWrite attempts to access a property name beginning with bshield, we simply append an underscore to the name, thus returning the property value that the original script would have seen. Since array indices can also be used to access object properties, we must return the appropriate value for the given index.

In the second case, the in construct allows iteration through all of an object's properties by name. The bshield property of an object must be hidden during the iteration if it is present. Thus, BrowserShield inserts a check as the first line of the iteration loop, jumping to the next item if the property name is bshield. This is accomplished using the rewrite rule shown in Table 1.

4.3 Transparency

The BrowserShield framework must also ensure its presence is transparent to the original script's semantics. The techniques for preventing tampering described in Section 4.2 contribute to this goal by making BrowserShield inaccessible. Transparency additionally requires that we present to scripts the context they would have in the absence of BrowserShield.

Shadow Copies Scripts can access both their own script code and HTML, which BrowserShield modifies for interposition. To preserve the intended semantics of such scripts, BrowserShield retains a "shadow copy" of all original code before rewriting it. The shadow copy is stored on a bshield property of the object. Interposition on property reads and writes allows the shadow copy to be exposed to scripts for access and modification.

Shadowing translated HTML requires additional care. During T_{HTML} transformation, a policy may rewrite static HTML elements. We must similarly create shadow copies for such translated HTML elements, but we cannot directly create a JavaScript object in HTML to store the shadow copy. Thus, we persist the shadow copy to a bshield HTML tag attribute during T_{HTML} , which is later used by the BrowserShield library. For example, a policy function that rewrites link URLs may modify the href attribute of <a>a> tags during the T_{HTML} transformation. Then, the persisted shadow copy looks like this:

```
<a href="http://translatedLink"
bshield="{href:'http://originalLink'}">
```

When BrowserShield looks for the bshield property of the DOM object corresponding to this tag, it interprets this string into an actual bshield property with a shadow copy for the href attribute.

Because scripts can only interact with shadow copies of their code and not modified copies, our transformations are not required to be idempotent. That is, we will never apply T_{HTML} or T_{script} to code that has already been transformed.

Preserving Context The JavaScript eval function evaluates a string as script code in the current scope, and any occurrence of the this keyword in the string is bound to the current enclosing object. Thus, if eval were to be called from within bshield.invokeFunc, the this keyword might evaluate differently than in the original context.

For this reason, the rewriting rule for functions is actually more complex than shown in Table 1. Instead, the rewritten code first checks if the function being invoked is eval. If so, the parameter is translated using T_{script} and then evaluated in the correct context; otherwise, <code>invokeFunc</code> is called as described before. Thus, the code is rewritten as follows:

```
bshield.isEval(bshield.func = foo) ?
  eval(bshield.translate(x)) :
  bshield.invokeFunc(bshield.func, x);
```

Note that the function expression foo is assigned to a temporary state variable on the bshield object, so that the expression is not evaluated a second time in the call to invokeFunc.

This check is a special case that is only needed for eval, because eval is the only native function in JavaScript that accesses this. Other native functions, such as alert or parseInt, do not access this, and can be evaluated within invokeFunc.

4.4 Flexible policies

The final goal of BrowserShield is to support flexible policy enforcement. This can be achieved by separating mechanism from policy: Our mechanism consists of the rewrite rules for translating HTML and script code, and our policy consists of the run-time checks invoked by the rewritten code. Some run-time checks are critical for complete interposition, such as applying T_{script} to any string passed to eval or the Function constructor, or applying T_{HTML} to any string passed to document.write or assigned to document.body.innerHTML. These checks are always applied, regardless of what policy is in place. Because the interposition is policy-driven, our system can be made incrementally complete. For example, if an undocumented API is discovered that can manipulate the document tree, we simply add a new policy to interpose on this API.

The remaining run-time checks are used for enforcing flexible policies, such as the MS04-040 vulnerability filter in Figure 2. Such policy functions are down-

loaded separately from the remainder of the Browser-Shield code, and they can be updated and customized based on the intended application.

Policy functions are given the chance to inspect and modify script behavior at all interposition points, including property reads and writes, function and method invocations, and object creations. We also allow policy writers to introduce new global state and functions as part of the global bshield object, or introduce local state and methods for all objects or for specific objects. Policy functions for HTML can also be registered by tag name. The tags are presented to HTML policy functions as part of a token stream of tags and text, without a full parse tree. It is also possible for policy functions to further parse the HTML token stream to gain additional context, although we have not yet encountered a need for this in the policies we have authored.

5 Implementation

We have implemented a prototype of BrowserShield as a service deployed at a firewall and proxy cache. Our prototype consists of a standard plugin to Microsoft's Internet Security and Acceleration (ISA) Server 2004 [21], and a JavaScript library that is sent to the client with transformed web documents. The ISA plugin plays the role of the BrowserShield logic injector.

We implemented our ISA plugin in C++ with 2,679 lines of code. Our JavaScript library has 3,493 lines (including comments). Most of the ISA plugin code is devoted to parsing HTML, while about half of the JavaScript library is devoted to parsing HTML or JavaScript. This is a significantly smaller amount of code than in a modern web browser, which implies that our trusted computing base is small compared to the code base we are protecting.

The ISA plugin is responsible for applying the T_{HTML} transformation to static HTML. The ISA plugin first inserts a reference to the BrowserShield JavaScript library into the web document. Because this library is distributed in a separate file, clients automatically cache it, reducing network traffic on later requests. T_{HTML} then rewrites all script elements such that they will be transformed using T_{script} at the client before they are executed. Figure 3 depicts this transformation; note that it does not require translating the JavaScript at the firewall.

In our implementation, the firewall component applies T_{HTML} using a streaming model, such that the ISA Server can begin sending transformed data to the client before the entire page is received. This streaming model also means that we do not expect the filter to be vulnerable to state-holding DoS attacks by malicious web pages.

One complexity is that BrowserShield's HTML parsing and JavaScript parsing must be consistent with that of the underlying browser. Any inconsistency will cause

false positives and false negatives in BrowserShield runtime checks. For our prototype, we have sought to match IE's behavior through testing and refinement. If future versions of browsers exposed this logic to other programs, it would make this problem trivial.

When the browser starts to run the script in the page, the library applies T_{script} to each piece of script code, translating it to call into the BrowserShield interposition layer. This may sometimes require decoding scripts, a procedure that is implemented in publicly available libraries [34] and which does not require cryptanalysis, though we have not yet incorporated it in our implementation.

A final issue in T_{script} is translating scripts that originate in source files linked to from a source tag. T_{HTML} rewrites such source URLs so that they are fetched through a proxy. The proxy wraps the scripts in the same way that script code embedded directly in the page is wrapped. For example, a script source URL of http://foo.com/script.js would be translated to http://rewritingProxy/translateJS.pl?url=http://foo.com/script.js. T_{script} is then applied at the client after the script source file is downloaded.

6 Evaluation

Our evaluation focuses on measuring BrowserShield's vulnerability coverage, the complexity of authoring vulnerability filters, the overhead of applying the BrowserShield transformations at firewalls, and the overhead of running the BrowserShield interposition layer and vulnerability filters at end hosts.

6.1 Vulnerability Coverage

We evaluated BrowserShield's ability to protect IE against all critical vulnerabilities for which Microsoft released patches in 2005 [1]. Of the 29 critical patches that year, 8 are for IE, corresponding to 19 IE vulnerabilities. These vulnerabilities fall into three classes: IE's handling of (i) HTML, script, or ActiveX components, (ii) HTTP, and (iii) images or other files. Table 2 shows how many vulnerabilities there were in each area, and whether BrowserShield or another technology could provide patch-equivalent protection. The BrowserShield design is focused on HTML, script, and ActiveX controls, and it can successfully handle all 12 of these vulnerabilities. This includes vulnerabilities where the underlying programmer error is at a higher layer of abstraction than a buffer overrun, e.g., a cross-domain scripting vulnerability. Handling HTTP accounted for 3 of the 19 vulnerabilities. Perhaps surprisingly, 2 out of 3 of these vulnerabilities required BrowserShield in addition to an existing HTTP filter, such as Snort [38] or Shield [43]. This is because malformed URLs could trigger the HTTP layer vulnerabilities regardless of whether the URL came over the network or was generated internally by the browser. BrowserShield is able to prevent the HTML/script layer from triggering the generation of these bad HTTP requests. Processing images or other files accounted for the remaining 4 vulnerabilities. Patch-equivalent protection for these vulnerabilities is already available using existing anti-virus solutions [13].

vulnerability		protected by			
type	#	BrowserShield	HTTP filter	antivirus	
HTML, script, ActiveX	12	12	0	0	
НТТР	3	2*	3*	0	
images and other files	4	0	0	4	

Table 2: BrowserShield Vulnerability Coverage. *Two of the HTTP vulnerabilities required both BrowserShield and an HTTP filter to provide patch-equivalent protection.

Because management and deployment costs are often incurred on a per-patch basis, we also analyze the vulnerabilities in Table 2 in terms of the corresponding patches. For the 8 IE patches released in 2005, combining BrowserShield with standard anti-virus and HTTP filtering would have provided patch-equivalent protection in every case, greatly reducing the costs associated with multiple patch deployments. In the absence of BrowserShield, anti-virus and HTTP filtering would have provided patch-equivalent protection for only 1 of the IE patches.

6.2 Authoring Vulnerability Filters

To evaluate the complexity of vulnerability filtering, we choose three vulnerabilities from three different classes: HTML Elements Vulnerability (MS04-040), COM Object Memory Corruption (MS05-037), and Mismatched DOM Object Memory Corruption (MS05-054).

We filtered for the MS04-040 vulnerability using the function shown in Figure 2. Registering this filter for each of the three vulnerable tags is as simple as:

bshield.addHTMLTagPolicy("IFRAME", func);

COM object vulnerabilities typically result from IE instantiating COM objects that have memory errors in their constructors. The IE patch blacklists particular COM objects (identified by their clsid). Implementing an equivalent blacklist requires adding checks for an HTML tag (the OBJECT tag) and sometimes a JavaScript function (the ActiveXObject constructor, which can be used to instantiate a subset of the COM objects accessible through the OBJECT tag). In the case of MS05-037, it does not appear to be possible to instantiate the vulnerable COM object using the ActiveXObject constructors.

tor. The OBJECT tag filter is conceptually similar to the function shown in Figure 2.

The MS05-054 vulnerability results when the window object, which is not a function, is called as a function in the outermost scope. Our interposition layer itself prevents window from being called as a function in the outermost scope since all function calls are mediated by BrowserShield with invokeFunc. Hence there is no need for a filter. Nevertheless, if this vulnerability had not depended on such a scoping constraint, we could simply have added a filter to prevent calling the object as a function.

To test the correctness of our vulnerability filters, we installed an unpatched image of Windows XP Pro within a virtual machine, and created web pages for each of the vulnerabilities that caused IE to crash. Applying BrowserShield with the filters caused IE to not crash upon viewing the malicious web pages. We tested the fidelity of our filters using the same set of URLs that we used in our evaluation of BrowserShield's overhead (details are in Section 6.3). Under side-by-side visual comparisons, we found that the filters had not changed the behavior of any of the web pages, as desired.

6.3 Firewall Performance

We evaluated BrowserShield's performance by scripting multiple IE clients to download web pages (and all their embedded objects) through an ISA server running the BrowserShield firewall plugin. The ISA firewall ran on a Compaq Evo PC containing a 1.7GHz Pentium 4 microprocessor and 1 GB RAM. Because we are within a corporate intranet, our ISA server connected to another HTTP proxy, not directly to web sites over the internet. We disabled caching at our ISA proxy, and we fixed our IE client cache to contain only the BrowserShield JavaScript library, consistent with the scenario of a firewall translating all web sites to contain a reference to this library.

We ran 10 IE processes concurrently using 10 pages that IE could render quickly (so as to increase the load on the firewall), and repeatedly initiated each page visit every 5 seconds. We used manual observation to determine when the load on the ISA server had reached a steady state

We chose these 10 pages out of a set of 70 URLs that are the basis for our client performance macrobenchmarks. This set is based on a sample of 250 of the top 1 million URLs clicked on after being returned as MSN Search results in Spring 2005, weighted by click-through count. Specifically, the 70 URLs are those that BrowserShield can currently render correctly; the remaining URLs in the sample encountered problems due to incomplete aspects of our implementation, such as JavaScript parsing bugs.

resource	unmodified	browsershield	
cpu utilization	15.0%	18.3%	
virtual memory	317 MB	319 MB	
working set	45.5 MB	46.6 MB	
private bytes	26.3 MB	27.3 MB	

Table 3: BrowserShield Firewall overheads. "Virtual memory" measures the total virtual memory allocated to the process; "working set" measures memory pages that are referenced regularly; "private bytes" measures memory pages that are not sharable.

We measured CPU and memory usage at the firewall, as shown in Table 3. CPU usage increased by about 22%, resulting a potential degradation of throughput by 18.1%; all aspects of memory usage we measured increased by negligible amounts. We also found that network usage increased only slightly (more detail in Section 6.4.2).

6.4 Client Performance

We evaluated the client component of our Browser-Shield implementation through microbenchmarks on the JavaScript interposition layer and macrobenchmarks on network load, client memory usage, and the latency of page rendering.

6.4.1 Microbenchmarks

We designed microbenchmarks to measure the overhead of individual JavaScript operations after translation. Table 4 lists our microbenchmarks and their respective BrowserShield slow-down. Our results are averages over 10 trials, where each trial evaluated its microbenchmark repeatedly, and lasted over half a second. For the first 11 micro-benchmarks, the standard deviation over the 10 trials was less than 2%. In the last case it was less than 8%. The slowdown ratio was computed using the average time required per microbenchmark evaluation with and without the interposition framework.

	operation	slowdown
1	i++	1.00
2	a = b + c	1.00
3	if	1.07
4	string concat ('+')	1.00
5	string concat ('concat')	61.9
6	string split ('split')	21.9
7	no-op function call	44.8
8	x.a = b (property write)	342
9	eval of minimal syntactic structure	47.3
10	eval of moderate syntactic structure, minimal computation	136
11	eval of moderate syntactic structure, significant computation	1.34
12	image swap	1.07

Table 4: BrowserShield Microbenchmarks. Slowdown is the ratio of the execution time of BrowserShield translated code and that of the original code.

Microbenchmarks 1-4 measure operations for which we expect no changes during rewriting, and hence no slowdown. The only slowdown we measure is in the case of the if statement. Further examination showed that the BrowserShield translation inserted a semi-colon (e.g., var a = 1 (linebreak) changed to var a = 1; (linebreak)). This results in a 7% slowdown.

Microbenchmarks 5-8 measure operations we expect to incur a slowdown comparable to an interpreter's slowdown. As detailed in Section 4, BrowserShield translation introduces additional logic around method calls, function calls, and property writes, leading to a slowdown in the range of 20x-400x. This slowdown is in line with good interpreters [32], but worse than what is achieved by rewriting systems targeting other languages, e.g., Java bytecode [10]. BrowserShield is paying a price for the JavaScript subtleties that previous rewriting systems did not have to deal with. We were curious about the difference in slowdown between the two string methods; an additional experiment showed that the difference can be attributed to the JavaScript built-in concat method requiring about 3 times as much CPU as the built-in split method. Also, it is not surprising that property writes have a greater slowdown than function or method calls because property writes need to both guard the BrowserShield namespace and interpose on writes to DOM elements (such as the text property of scripts).

Microbenchmarks 9-11 explore the overhead of translating JavaScript code of various complexity. The "eval of minimal syntactic structure" microbenchmark measures the cost of translating and then evaluating a simple assignment. The cause of the large slowdown is the additional work done by eval in the BrowserShield framework: parsing, constructing an AST, modifying the AST, and outputting the new AST as a JavaScript program. The two subsequent "eval of moderate syntactic structure" microbenchmarks measure the cost of translating and evaluating a simple for (;;) loop. This simply demonstrates that as the cost of the computation inside the simple loop increases, the cost of translating the code can decrease to a small fraction of the overall computational cost.

The last microbenchmark measures the overhead of performing a simple manipulation of the DOM – swapping two 35 KB images. This microbenchmark is designed to measure the relative importance of overheads in the JavaScript engine when the JavaScript is manipulating the layout of the HTML page. The JavaScript code to swap these two images requires two property writes (i.e., img.src = 'newLink'), and we described above how BrowserShield translation adds significant overhead to property writes. Nonetheless, the overall slowdown is less than 8%. In particular, the raw time to swap the image only increases from 26.7 milliseconds to 28.5 milliseconds. This suggests that even the large overheads

that BrowserShield translation adds to some language constructs may still be quite small in the context of a complete web page.

In summary, BrowserShield incurs a significant overhead on the language constructs where it must add interpreter-like logic, but these overheads can be quite small within the context of the larger DOM manipulations in embedded scripts.

6.4.2 Macrobenchmarks

We designed macrobenchmarks to measure the overall client experience when the BrowserShield framework is in place. In particular, the macrobenchmarks include all the dynamic parsing and translation that occurs before the page is rendered, while the microbenchmarks primarily evaluated the performance of the translated code accomplishing a task relative to the untranslated code accomplishing that same task. To this end, we scripted an instance of IE to download each of the 70 web pages in our workload 10 times. For the same reasons given in our evaluation of the BrowserShield ISA component, we maintained that the only object in the IE cache was the BrowserShield JavaScript library. These caching policies represent a worst-case for client latency. This measurement includes the overhead of the three filters that we discussed in Section 6.1. We then repeated these measurements without the BrowserShield framework and translation.

We set a 30 second upper limit on the time to render the web page, including launching secondary (popup) windows and displaying embedded objects, but not waiting for secondary windows to render. We visually verified that the programmatic signal that rendering had completed indeed corresponded to the user's perception that the page had rendered. IE hit the 30-second timeout several times in these trials, and it hit the timeouts both when the BrowserShield framework and translation were present and when the framework and translation were absent. We did not discern any pattern in these timeouts, and because our experiments include factors outside our control, such as the wide-area network and the servers originating the content, we do not expect page download times to be constant over our trials. We re-ran the trials that experienced the timeouts.

Figure 5 shows the CDF of page rendering with and without BrowserShield. On average BrowserShield added 1.7 seconds to page rendering time. By way of contrast, the standard deviation in rendering time without BrowserShield was 1.0 seconds.

In Figure 6, we further break down the latency for the 10 pages that took the most time to render under BrowserShield. They experienced an average increase in latency of 6.3 seconds, requiring 3.9 seconds on average without BrowserShield and 10.2 seconds on average with

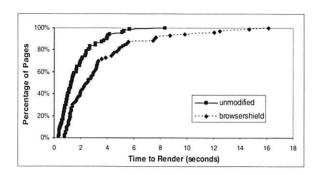


Figure 5: Latency CDF with and without BrowserShield

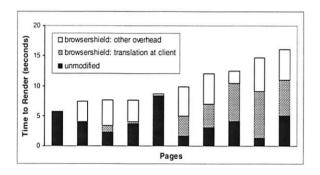


Figure 6: Breakdown of latency for slowest 10 pages under BrowserShield

BrowserShield. Of this 6.3 seconds of increased latency, we found that 2.8 seconds (45%) could be attributed to the overhead of dynamically translating JavaScript and HTML within IE. We attribute the remaining overhead to effects such as the overhead of evaluating the translated code, and the time to modify the HTML at the firewall.

We broke down the latency of dynamic translation for both HTML and JavaScript into 2 parts each: time to parse the JavaScript/HTML into an AST and convert the modified AST back to a string, and the time to modify the AST. We found that the time to parse the JavaScript to and from a string was always more than 70% of the overall latency of dynamic translation, and it averaged 80% of the overall latency. Figure 7 shows the JavaScript parsing time versus the number of kilobytes. Fitting a least-squares line to this data yields an average parse rate of 4.1 KB of JavaScript per second, but there was significant variation; the slowest parse rate we observed was 1.3 KB/second.

Figure 8 shows the memory usage of page rendering with and without BrowserShield. We found that private bytes (memory pages that are not sharable) was the client memory metric that increased the most when rendering the transformed page. Private memory usage increased on average by 11.8%, from 19.8 MB to 22.1 MB. This increase was quite consistent; no page caused memory usage to increase by more than 3 MB.

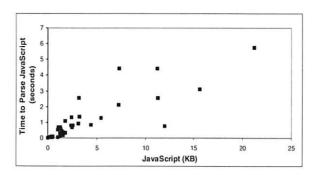


Figure 7: Latency of JavaScript parsing

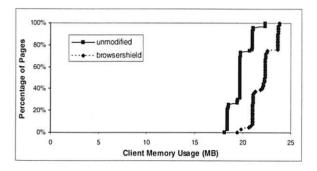


Figure 8: Memory Usage at Client

We also measured the increased network load over a single run through the pages both with and without BrowserShield. We measured an average increase of 9 KB, less than the standard deviation in the network load over any individual trial due to background traffic during our measurements. We expect BrowserShield rewriting to only slightly increase the network load, because the firewall just adds script wrappers, while the translation itself happens at the client.

7 Related Work

We first compare with other protection systems in Section 7.1. We then discuss BrowserShield's relation to the extensive work on code rewriting and interposition in Section 7.2.

7.1 Remote Exploit Defense

In our prior work on Shield [43], we proposed using vulnerability-specific filters to identify and block network traffic that would exploit known software vulnerabilities. Shield maintains protocol-specific state machines in an end-host's network stack, allowing it to recognize when a packet will trigger a vulnerability. However, the Shield approach does not address dynamic content such as scripts in web documents, since it is undecidable whether script code in a document will eventually exploit a vulnerability. BrowserShield shares Shield's focus on vulnerability-specific filters, but in contrast to

Shield, its use of runtime interposition allows it to handle exploits in dynamic HTML.

Like BrowserShield and Shield, IntroVirt also employs vulnerability-specific predicates, specifically to detect past and present intrusions using virtual machine introspection and replay [23]. As a result, IntroVirt allows "just in time" patch application: postponing the application of a patch while an exploit has not occurred, and rolling back system state to apply a patch if an exploit does occur. BrowserShield instead offers protection while a patch is being tested (or otherwise delayed) by the administrator of a vulnerable system, buying time even in cases where exploits are immediately attempted. Additionally, BrowserShield supports more flexible deployment scenarios. For example, it does not require the client's browser to run inside an instrumented virtual machine.

Opus [2] seeks to address the problem of patching by allowing patches to be applied without restarting the application. Opus provides tools for developers to increase the reliability of such "dynamic" patches during development. However, these tools reduce, but do not eliminate, the programmer's burden to produce a correct dynamic patch.

Vigilante [7] focuses on worm containment, automatically detecting and preventing the spread of worm traffic. Vigilante combines rapid distribution of self-certifying alerts and automatic filter generation, along with vulnerability detection techniques such as non-executable pages [30] and dynamic dataflow analysis [29]. These techniques, even with the Vigilante improvements, admit false negatives. BrowserShield does not share the speed constraint of Vigilante, since browser exploits require human involvement and therefore do not spread on the same time scales as worms. Therefore, we are able to trade off the speed of automatically generated vulnerability filters for the accuracy of hand-coded filters.

EarlyBird [36] and Autograph [24] are two exemplary systems that use pattern matching to block network traffic containing exploits. Pattern matching scales to high data rates, crucial to the authors' goal of stopping worm outbreaks at network choke points. The HTML scripts that are BrowserShield's focus seem difficult to detect consistently with pattern matching, as they can trivially modify themselves at the client.

HoneyMonkey [44] aims to discover web servers that distribute malicious code. In HoneyMonkey, virtual machines automatically fetch content from the web and use black-box techniques to discover exploits. Exploit discovery is complimentary to BrowserShield's approach of providing patch-equivalent protection to clients.

Finally, a number of techniques have aimed to sandbox the browser or other applications, in effect protecting the operating system from the impact of an exploit. These techniques include system call interposition [17, 18, 22] and Microsoft's "protected mode" for IE in Windows Vista [20]. These may limit damage to a user's computing environment, but they do not protect the browser itself, allowing attacks such as keylogging to easily be conducted from exploited browsers. Tahoma [8] takes the confinement approach one step further, sandboxing browsers in virtual machines and using site-specific manifests to restrict browser traffic to known servers. While this could help to mitigate many browser related problems, the difficulty of getting such manifests widely adopted is unclear.

7.2 Interposition Techniques

Interposition techniques such as code rewriting have been used in previous work to achieve additional safety properties or otherwise modify the behavior of existing code. Code rewriting is only one of several alternatives for backward compatible modifications, and the choice of technique is influenced by tradeoffs in deployability and performance. Directly modifying the execution environment, such as the Java Virtual Machine, has the highest deployment barriers. Some work instead uses a level of indirection, such as emulation (e.g, Bochs [6]), easing deployment but incurring a high performance overhead. Thus, BrowserShield and others [10, 41, 42] employ code rewriting, with its low barriers to deployment and smaller performance overhead than that required by an emulator.

We characterize interposition techniques by the target of interposition, since the technical differences between targets require different solutions. Compared to approaches for other interposition targets, BrowserShield must address a new combination of technical challenges presented by JavaScript: its scoping rules, an implicit garbage collector, pervasive reflection, and its prototype-based object model (which implies a lack of static typing).

Machine Code Many approaches focus on the machine code interface, whether rewriting binary instructions or emulating them at runtime. Software Fault Isolation (SFI) [41] rewrites binary code to insert runtime checks, creating sandboxes that prevent code from writing or jumping to addresses outside its fault domain. This creates process-like memory boundaries between units of code within a process. The more recent XFI [9] uses binary rewriting to provide flexible access control and additional integrity guarantees. VMware ESX Server [42] also rewrites machine code, in its case to allow programs to be virtualized on x86 hardware. Etch [31] rewrites machine code with the goals of profiling and measurement. Valgrind [40] and Program Shepherding [25] are dynamic binary instrumentation tools. Valgrind's goal is

to offer debugging and profiling support, while Program Shepherding's goal is to monitor control flow, preventing the transfer of control to data regions which might include malicious code.

The techniques used for rewriting at the machine code interface do not need to address any of the four challenges of JavaScript rewriting that have influenced BrowserShield: scoping, reflection, garbage collection or typing. Most work interposing at the machine code interface only adds semantics that can be defined in terms of low level operations, such as enforcing a process-like memory boundary, as in SFI. Indeed, Erlingsson and Schneider [11] note the difficulty of extending rewriting at the machine code interface to enforce policies on the abstractions internal to an application. BrowserShield's interposition target (the HTML document tree) is such an application-internal abstraction.

System Call Interface Much previous work has modified user level program behavior by interposing on the system call interface. Jones introduces a toolkit for system call interposition agents that simplifies tasks such as tracing, emulation, and sandboxing [22]. Wagner et al. use system call interposition in Janus to confine untrusted applications to a secure sandbox environment [18]. Garfinkel notes difficulties in trying to interpose on the system call interface [16], such as violating OS semantics, side effects, and overlooking indirect paths. Garfinkel et al. discuss a delegation-based architecture to address some of these problems [17]. Naccio describes an approach to provide similar guarantees by rewriting x86 code that links against the Win32 system call interface [12]. Naccio can also rewrite Java bytecode.

Work on the system call interface differs from BrowserShield both in goal and in technique. System call interposition can guard external resources from an application, while the goal of BrowserShield is to guard an application-internal resource, the HTML document tree. Naccio's use of rewriting as a technique to interpose on the system call interface does not present any of the four technical challenges (scoping, reflection, garbage collection or typing) relevant to JavaScript rewriting. For example, Naccio also wraps methods to accomplish interposition, but Naccio's method wrappers do not need to handle JavaScript's scoping rules, and so do not need to implement swapping.

Java Bytecode Several pieces of previous work [10, 11, 37], including the previously mentioned Naccio [12], have used rewriting at the Java Virtual Machine bytecode interface [26]. This interface is type-safe, and provides good support for reasoning about application-internal abstractions. In the most similar of these works to BrowserShield, Erlingsson's PoET mechanism rewrites

Java bytecode to enforce security policies expressed in the PSLang language [10].

JavaScript's pervasive reflection, scoping rules, and prototype-based object model forced us to develop several techniques not needed for Java bytecode rewriting. For example, where Java bytecode rewriting can interpose on Java's reflection API, BrowserShield must interpose on all property reads and writes, as well as some for loops, to achieve similar control over reflection. Additionally, Java bytecode rewriting can achieve complete interposition by only modifying callees (using method wrappers) and without maintaining state, though some previous work allowed modifying callers or adding state to simplify policy construction [10]. In contrast, BrowserShield must modify both callers and callees to appropriately handle scoping and the possibility of functions aliased as methods (and vice versa). Also, Browser-Shield must maintain state, requiring careful attention to its interaction with the JavaScript garbage collector.

Web Scripting Languages We are not aware of any full interposition techniques for web scripting languages like JavaScript. The SafeWeb anonymity service used a JavaScript rewriting engine that failed to provide either complete interposition or transparency [27]. The Greasemonkey [19] extension to the Firefox browser allows users to run additional site-specific scripts when a document is loaded, but it does not provide complete interposition between existing script code and the HTML document tree.

8 Conclusion

Web browser vulnerabilities have become a popular vector of attacks. Filtering exploits of these vulnerabilities is made challenging by the dynamic nature of web content. We have presented BrowserShield, a general framework that rewrites HTML pages and any embedded scripts to enforce policies on run-time behavior. We have designed BrowserShield to provide complete interposition over the underlying resource (the HTML document tree) and to be transparent and tamper-proof. Because BrowserShield transforms content rather than browsers, it supports deployment at clients, firewalls, or web publishers. Our evaluation shows that adding this approach to existing firewall and anti-virus techniques increases the fraction of IE patches from 2005 that can be protected at the network level from 12.5% to 100%, and that this protection can be done with only moderate overhead.

We have focused on the application of vulnerability-driven filtering in this paper, but JavaScript rewriting techniques may also enable new functionality for AJAX (Asynchronous JavaScript and XML) applications. Some potential uses include: eliminating the effort currently required to modify a website for the Coral [15]

CDN; modifying the cached search results returned by web search engines to redirect links back into the cache (since the original site may be unavailable); allowing appropriately sandboxed dynamic third-party content on a community site (such as a blog or wiki) that currently must restrict third-party content to be static; and debugging JavaScript code when attaching a debugger is infeasible, perhaps offering call traces or breakpoint functionality for complex scripts. User interface changes could even be added to make phishing more difficult, e.g., enforcing the display of origin URLs on all popup windows. As this list suggests, we are optimistic that JavaScript rewriting is a widely applicable technique.

References

- Microsoft Security Bulletin Summaries and Webcasts, 2005. http://www.microsoft.com/technet/security/bulletin/ summary.mspx.
- [2] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online Patches and Updates for Security. In *Usenix Security*, 2005.
- [3] J. P. Anderson. Computer Security Technology Planning Study Volume II. ESD-TR-73-51, Vol. II, Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford, MA, October 1972.
- [4] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of Vulnerability: a Case Study Analysis. *IEEE Computer*, December 2000.
- [5] S. Beattie, S. Arnold, C. Cowan, P. Wagle, and C. Wright. Timing the Application of Security Patches for Optimal Uptime. In LISA, 2002.
- [6] Bochs. http://bochs.sourceforge.net/.
- [7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In SOSP, 2004.
- [8] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A Safety-Oriented Platform for Web Applications. In *IEEE Symposium on Security and Privacy*, 2006.
- [9] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In OSDI, 2006.
- [10] Ú. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, 2000.
- [11] Ú. Erlingsson and F. B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In WNSP: New Security Paradigms Workshop, 2000.
- [12] D. Evans and A. Twyman. Flexible Policy-Directed Code Safety. In *IEEE Symposium on Security and Privacy*, 1999.
- [13] Eweek: Anti-Virus Protection for WMF Flaw, December 2005. http://www.eweek.com/article2/0,1895,1907102,00.asp.
- [14] Mozilla Security Alerts and Announcements. http://www.mozilla.org/security/.
- [15] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing Content Publication with Coral. In NSDI, 2004.
- [16] T. Garfinkel. Traps and Pitfalls: Practical Problems in in System Call Interposition based Security Tools. In NDSS, 2003.
- [17] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In NDSS, 2004.

- [18] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Usenix Security*, 1996.
- [19] Greasemonkey. http://greasemonkey.mozdev.org/.
- [20] Protected Mode in Vista IE7. http://blogs.msdn.com/ie/archive/2006/02/09/528963.aspx.
- [21] ISA Server. http://www.microsoft.com/isaserver/default.mspx.
- [22] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In SOSP, 1993.
- [23] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions Through Vulnerability-specific Predicates. In SOSP, 2005.
- [24] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Usenix Security*, 2004.
- [25] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution via Program Shepherding. In *Usenix Security*, 2002.
- [26] T. Lindholm and F. Yellin. The Java Virtual Machine Specification, 2nd edition, 1999.
- [27] D. Martin and A. Schulman. Deanonymizing Users of the SafeWeb Anonymizing Service. In USENIX Security, 2002.
- [28] Microsoft Security Bulletin MS04-040, December 2004. http://www.microsoft.com/technet/security/Bulletin/MS04-040.mspx.
- [29] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In NDSS, 2005.
- [30] Pax. http://pax.grsecurity.net/.
- [31] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Usenix NT Workshop*, 1997.
- [32] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. The Structure and Performance of Interpreters. In ASPLOS, 1996.
- [33] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In SOSP, 1973.
- [34] Windows Script Decoder. http://www.virtualconspiracy.com.
- [35] Secure Computing. http://www.securecomputing.com/pdf/WW-SSLscan-PO.pdf.
- [36] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In OSDI, 2004.
- [37] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In SOSP, 1999.
- [38] The Open Source Network Intrusion Detection System. http://www.snort.org/.
- [39] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In OOPSIA, 1987.
- [40] Valgrind. http://www.valgrind.org/.
- [41] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In SOSP, 1993.
- [42] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In OSDI, 2002.
- [43] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In SIGCOMM, 2004.
- [44] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In NDSS, 2006.

XFI: Software Guards for System Address Spaces

Úlfar Erlingsson Microsoft Research Silicon Valley

Martín Abadi Microsoft Research Silicon Valley & UC Santa Cruz Michael Vrable UC San Diego

Mihai Budiu Microsoft Research Silicon Valley George C. Necula UC Berkeley

Abstract

XFI is a comprehensive protection system that offers both flexible access control and fundamental integrity guarantees, at any privilege level and even for legacy code in commodity systems. For this purpose, XFI combines static analysis with inline software guards and a two-stack execution model. We have implemented XFI for Windows on the x86 architecture using binary rewriting and a simple, stand-alone verifier; the implementation's correctness depends on the verifier, but not on the rewriter. We have applied XFI to software such as device drivers and multimedia codecs. The resulting modules function safely within both kernel and user-mode address spaces, with only modest enforcement overheads.

1 Introduction

XFI is a comprehensive software protection system that supports fine-grained memory access control and fundamental integrity guarantees for system state. XFI offers a flexible, generalized form of software-based fault isolation (SFI) [25, 36, 41] by building on control-flow integrity (CFI) [1, 2] at the machine-code level. This CFI foundation enforces external and internal interfaces, enables efficient XFI mechanisms, and helps protect the integrity of critical state, such as the x86 control registers.

In comparison with other protection alternatives, XFI requires neither hardware support [39, 44] nor type-safe programming languages [5, 19, 24]. XFI does not restrict memory layout and is compatible with system aspects such as signals and multi-threading. Furthermore, XFI applies at any privilege level, and even to legacy code that is run natively in the most privileged ring of x86 systems; in this respect, we regard XFI as achieving an important practical goal.

XFI has a clear architecture, whose basic implementation can be relatively straightforward and trustworthy. XFI protection is established through a combination of

static analysis with inline software *guards* that (much as in SFI) perform checks at runtime. The *XFI verifier* performs the static analysis as a linear inspection of the structure of machine-code binaries; it ensures that all execution paths contain sufficient guards before any possible protection violation. Verification is simple and, in principle, amenable to formal analysis and other means of assuring correctness. An *XFI module* is an executable binary that passes verification; such modules can be created by hand, by compile-time code generation, or by binary rewriting. However, software that hosts XFI modules need trust only the verifier, not the means of module creation. Thus, XFI modules can be seen as an example of proof-carrying code (PCC) [29], even though they do not include logical proofs.

XFI protection relies on several distinct runtime mechanisms, whose correct use is established by the XFI verifier. Guards ensure that control flows only as expected, even on computed transfers, and similarly that memory is accessed only as expected. Multiple memory accesses can be checked by a single memory-range guard, optimized for fast access to the most-frequently-used memory. XFI also employs two stacks. The regular execution stack provides a scoped stack, which holds data accessible only in the static scope of each function, including return addresses and most local variables. The scoped stack cannot be accessed via computed memory references, such as pointers; therefore, it serves as isolated storage for function-local virtual registers. A separate allocation stack holds other stack data which may be shared within an XFI module. Like heap memory, the allocation stack may be corrupted by buffer overflows and other pointer errors.

XFI protection can be of benefit to any *host system* that loads binary modules into its address space to make use of their functionality. Operating systems are example host systems, as are web browsers. Conversely, those modules may rely on their host system, by invoking its

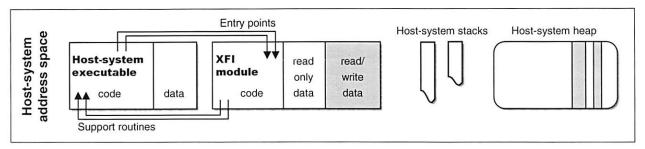


Figure 1: The address space of a host system with an XFI module. The module's external interfaces are restricted, as shown by the arrows. When the module uses a host-system stack, it is as a protected, scoped stack of virtual registers. Shaded areas are subject to arbitrary writes by the XFI module. Optionally, the module may read all memory.

support routines. For example, the XFI module in Figure 1 can use two support routines, and can be called through two *entry points* by its host system. The module can read its own code and constants (or, optionally, all of memory). It may read and write a section of its loaded module binary, and two contiguous heap regions to which the host system has granted it write access. It cannot modify its own code at runtime, so it cannot invalidate the properties established by the verifier statically.

XFI requires no complex infrastructure from host systems: XFI modules are normal executable binaries (e.g., DLLs), and can be loaded and used as such. As a result, host systems may sometimes be unaware of whether or not they are using an XFI module. In general, however, XFI protection does rely on support components, for instance for thread-specific management and for granting and revoking memory-access permissions.

We have implemented XFI for Windows on the x86 architecture as two main components: first, a relatively complex XFI rewriter, based on Vulcan [37], that instruments and structures executable binaries, and, second, a smaller, self-contained verifier. We have also designed and built host-system support components. Further, we have studied optional, lightweight hardware support for XFI, through cycle-accurate simulations.

We have used our implementation for creating independently verifiable dynamic libraries, device drivers, and multimedia codecs. The resulting XFI modules function safely within the address spaces of the Windows kernel and the Internet Explorer web browser, and provide services to those host systems. Thus, while still only a prototype, our implementation is already practical.

Although we have not explored many optimizations, XFI enforcement overhead is modest: in our experiments it ranged from 5% to a factor of two, on current x86 hardware. This overhead is acceptable for many systems; sometimes it can be lower than that of traditional hardware-supported address spaces (because context switching across XFI modules is fast [27, 41]).

Sections 2 and 3 describe XFI policies and mechanisms, respectively. Section 4 presents our implementa-

tion of these mechanisms. Section 5 discusses the application of XFI protection to device drivers and multimedia codecs. Section 6 gives measurements and other results of our experiments. Section 7 puts XFI in the context of related work, and Section 8 concludes.

2 Policies

XFI guarantees that certain properties hold during the execution of XFI modules. Some of these properties are fundamental ground rules about the interaction between XFI modules and their system environment. Others are convenient, auxiliary rules for this interaction. Yet others deal with the internal behavior of XFI modules, and contribute to security.

In discussing the implications of these properties, we rely on a running example: the Independent JPEG Group's image-decoding reference implementation [20]. This JPEG codec is an important, ubiquitous, legacy program. It is intended to behave as a pure function that reads compressed image data and writes an output bitmap. We have turned this codec into an XFI module, and used that module in the Internet Explorer web browser; we can equally well use it in the Windows kernel. XFI protection implies that image decoding does behave as a side-effect-free procedure.

2.1 External Properties

An XFI module complies with the following policy in its interactions with its system environment:

- P1 Memory-access constraints: Memory accesses are either into (a) the memory of the XFI module, such as its global variables, or (b) into contiguous memory regions to which the host system has explicitly granted access. In both cases, read, write, and execute operations are handled separately, and may access different regions. In particular, no writes are permitted into the code of the XFI module.
- P2 Interface restrictions: Control can never flow outside the module's code, except via calls to a set of prescribed support routines, and via returns to external call-sites.

P3 Scoped-stack integrity: The scoped stack is always well formed. In particular: (a) the stack register points to at least a fixed amount of writable stack memory; (b) the stack accurately reflects function calls, returns, and exceptions; (c) the Windows stack exception frames are well formed, and linked to each other.

P1 and P3 aim to ensure that state has appropriate values. In general, for memory, we do not know what those values are, so we trust particular components to maintain those values properly—hence we need to authenticate and authorize accesses to memory, as indicated in P1. For the scoped stack, on the other hand, we can formulate integrity properties, and preserve them as invariants, as indicated in P3. As for P2, it addresses external interface protection. Much as in other protection systems [35], P2 does not, by itself, rule out the improper use of legitimate interfaces, for example calling functions in an unsafe order. However, XFI facilitates checking compliance with safety specifications by other means (see Section 2.3).

With XFI, memory and interfaces can be protected at the granularity of individual bytes and calls, respectively. Moreover, different parts of an XFI module may enjoy different access rights, and—via special guards—XFI can support write-once, call-once, and other elaborate policies. For example, our JPEG module can be given access to the exact regions of an image data and an output bitmap (respectively, read-only and write-only), no matter where this memory resides. Moreover, although the JPEG module currently needs no external support, it could be safely allowed to invoke support routines, for instance to output status or error messages.

P1, P2, and P3 are more restrictive than strictly necessary. For example, an XFI module might safely be allowed to overwrite certain constants within its code section or to call other XFI modules instead of just host-system support routines. However, in what follows, we do not relax these properties but rather strengthen them.

Our additional restrictions do not, in practice, hinder the use of XFI protection for existing software; they do, however, generally help our implementation. In particular, the following restrictions P4 and P5 imply that, on the x86, we need not consider instructions with esoteric external effects on segments, descriptors, or page tables. Thus, P4 and P5 help ensure that an XFI implementation is complete and correct, despite complexities of the hardware architecture.

P4 Simplified instruction semantics: Certain machinecode instructions can never be executed. These include dangerous, privileged instructions (e.g., modifying x86 task descriptors), as well as potentially harmless but deprecated instructions (e.g., far jumps between segments). Certain other machine-code in-

- structions may be executed only in a context that constrains their effects; this context may be established by static verification, or by a specific dynamic check.
- P5 System-environment integrity: Certain aspects of the system environment, such as the machine model, are subject to invariants. For instance, the x86 segment registers cannot be modified, nor can the x86 flags register—except for condition flags, which contain results of comparisons.

In particular, these properties allow the x86 flags register to be safely saved on the scoped stack. They also allow the safe use of x86 programmed-I/O instructions, via a guard that restricts dynamically the I/O ports used. The stack-register integrity of P3(a) is a special case of P5.

System-environment integrity is a key advantage and differentiator of XFI protection. It is especially crucial to highly privileged host systems, such as operating system kernels. Without this protection, any computation, when run at high privilege, can fatally corrupt system state in a number of ways—not only through an incorrect, stray write to memory. In particular, XFI can restrict dangerous operations, such as modification of the x86 flags register, so that they are safe. For example, even if our JPEG module allowed buffer overflows, it could never tamper with the flag that controls the direction of REP memory operations. Thus, XFI can eliminate the possibility of unexpected ring changes, x86 "double faults", and other fatal conditions.

2.2 Internal Properties

In addition to restricting interactions between a module and its host, XFI places constraints on the execution of the module:

- P6 Control-flow integrity: Execution must follow a static, expected control-flow graph, even on computed calls and jumps. In particular, function calls must target the start of functions, and those functions must return to their callers.
- P7 Program-data integrity: Certain module-global and function-local variables can be accessed only via static references from the proper instructions in the XFI module. These variables are not subject to computed memory access.

P6 and P7 enable a simple, practical XFI verifier. We would find it much harder to have P1–P5 without P6 and P7. In particular, P6 prevents jumps that could circumvent guards, and P7 serves for securing temporary state. On the x86 architecture, P6 also supports P4 by preventing jumps into the middle of an instruction, where another, forbidden instruction may be encoded.

Furthermore, XFI protection constrains software internals in ways that make many attacks impossible. In com-

bination with P1-P5, P6 implies that attackers cannot subvert control flow, and P7 implies that attackers cannot overwrite certain program state, such as global variables that hold configuration data or function-local variables that indicate authentication status. These properties offer a strong defense against actual attacks (see [2, 12, 32]). Although an XFI module may still allow certain buffer overflows and other vulnerabilities, XFI protection can prevent exploits of those vulnerabilities. In particular, XFI guarantees correct function returns (cf. [13]), and prevents introduction of new machine code as well as unexpected uses of existing code. Thereby, XFI foils two of the most popular exploit techniques: code injection and jump-to-libc attacks [32]. For example, our JPEG module cannot be exploited by any of the several published JPEG attacks [2], despite originating in the same source code as vulnerable versions.

2.3 Additional Properties

P6 and P7 can also serve as a foundation for the dynamic enforcement of additional properties, such as the validity of system-call arguments and (more generally) compliance with specifications for the use of interfaces. P6 implies that dynamic checks cannot be circumvented, while P7 provides secure storage for state used by those checks.

Going further, P1–P7 support the enforcement of many other useful properties. We have implemented the following one, because it provides a guarantee that is important in our application of XFI to Windows device drivers, described in Section 5.

Assured self-authentication: An XFI module authenticates itself to the host system, e.g., by passing an immutable value chosen by the host system as an extra argument to support routines.

We have considered several other properties (without yet finding the need to realize them). For instance, when it is difficult or awkward to implement timeouts (e.g., in a kernel interrupt handler), XFI could bound the length of execution before a support routine is invoked. Alternatively, XFI could allow interrupts to be disabled when they are re-enabled within a fixed number of instructions.

3 Mechanisms

Several mechanisms can enforce the policies of Section 2. We have chosen mechanisms that allow for an x86 software implementation with good runtime performance. However, our mechanisms should apply to any architecture, and could be combined with hardware-based means of protection.

In our choice of mechanisms, we have considered not only efficiency but also trustworthiness. Because of restrictions such as P4, our mechanisms safely exclude problematic aspects of the x86 architecture. Also, complex rewriters and compilers are not in our trusted com-

puting base. Crucially, correctness depends on an independent static verifier. In principle, correctness could be established formally by analysis of this verifier. We have created proofs for CFI for a small machine-code language [1]; in further work, those proofs could be extended to the full XFI system given a partial formal model of the x86 instruction set (e.g., as in [3, 11, 25]).

This section describes the XFI mechanisms and outlines their implementation. The next section provides further details.

Static Code Inspection The XFI verifier checks statically that each XFI module has the appropriate structure and the necessary guards. For instance, the verifier checks that the XFI module includes only certain instructions. The guards should ensure that, when executed on the intended host system, the module will satisfy sufficient conditions for properties P1–P7 and, optionally, other requirements specific to the host system.

In particular, the verifier establishes constraints on control flow and memory accesses. For direct branches and jumps that statically meet those constraints, no CFI guard is required. Similarly, for direct access to global variables in the XFI module and to virtual registers on the scoped stack, no memory-range guard is required. In other cases, the verifier ascertains the presence of proper guards; these guards must precede the instruction in question along all execution paths.

XFI verification can be done by the users of extensible systems, in order to establish the safety of an XFI module before it is installed or executed. Optionally, verification could be repeated at runtime, for example to counter the corruption of read-only memory by an external device. Verification may be useful in other scenarios as well. For instance, for systems structured as distinct modules with clearly defined interactions (e.g., with COM [7]), verification may be a quality-assurance step during software production.

Section 4.3 gives further details on the verifier.

CFI Guards on Computed Control-Flow Transfers As indicated above, computed control-flow instructions may be preceded by CFI guards that ensure control transfers are within the allowed control-flow graph.

In one implementation [2], a fresh, constant identifier is assigned to sources and targets of computed control-flow transfers. An identifier is embedded within the code of a guard at each source and immediately before the code of each target. At runtime, the guard compares the identifier of the source and target, and ensures that they match. The verifier knows the identifiers and ensures that they appear in the code segment only at valid targets.

Figure 2 shows machine code for a guard. (For clarity, this example, and most other machine-code in this paper, is written in a generic, 32-bit assembly notation with x86

Figure 2: A computed call instruction, with a CFI guard and one valid callee destination.

register names.) This guard checks that right before the target destination EBX is the four-byte constant identifier 0×12345678 . The guard embeds that constant minus one as a literal, not the constant itself, so that the guard's code bytes do not become a valid target. Not shown is the guard that ensures that EBX points to XFI module code.

A guard is needed for computed jumps that may otherwise target unexpected addresses—because of error or attack—but not for function returns, since function-return addresses are kept safe in virtual registers (described below).

Two Stacks XFI makes use of two stacks: the regular execution stack for its scoped stack, as well as a separate allocation stack. This separation prevents the corruption of values, such as return addresses, that are subject to integrity guarantees. The scoped stack is used only in a stylized, structured manner, and is never the target of a computed memory access; therefore the integrity of values on the scoped stack can be established by the XFI verifier. The allocation stack is used for those stack values that an XFI module may access via a pointer. These include all local variables whose address is taken (e.g., arrays, or variables passed as call-by-reference arguments to functions).

At runtime, guards ensure that the two stacks do not overflow, and, thereby, that property P3(a) holds. Such guards must be placed at each module entry point, as well as in any cycle in the module's function-call graph. (Similar guards have been used for implementing noncontiguous stacks for lightweight threads [40].)

Statically, all explicit references to the scoped stack must be to SSP+K, where K is a positive, properly aligned constant, and SSP is the scoped-stack pointer register. (On the x86, SSP is the regular x86 stack register ESP, so that pushes, pops, calls, and returns use the scoped stack.) Furthermore, at each instruction of a module function, the scoped stack must be of a known, constant depth. Within each function, SSP can be lowered and raised only by constant amounts, in a controlled fashion. SSP is preserved by function calls.

The memory of the scoped stack is not accessible to an XFI module, except in the manner described above. Therefore, locations on the scoped stack can function as virtual registers: like registers, they are thread-local state that is only accessed by name. In realizing property P7, these virtual registers can hold local function variables and other XFI module state to be protected from memory corruption within the module.

On the other hand, the allocation stack is used for thread-local module data that is accessible via pointers (and, thus, potentially vulnerable to pointer errors). The allocation stack implements an efficient alternative to the heap allocation of this data; it is indexed by the allocation-stack pointer (ASP) register. (ASP can be a virtual register or, on the x86, the register EBP.) Within each function, ASP can be modified only in a controlled fashion, by either a constant amount or one bounded by a runtime check. ASP is preserved by function calls.

XFI requires only the scoped stack for certain simple modules, including those where arrays and pointer-accessible variables reside only on the heap. A host system that loads only such modules may omit all support for allocation stacks. In general, however, host systems must support this two-stack execution model, in particular by managing allocation stacks and by properly passing arguments and results between stacks. Section 4.5 describes the components of this support.

Guards on Computed Memory Accesses Whether it uses an allocation stack or not, a module must be able to access the memory that contains its code, read-only constants, and writable global variables, in order to execute, read, and read or write that memory, respectively. In addition, a host system may wish to give selective access to other memory regions. XFI uses guards for enabling the use of those additional regions—in any number, at any granularity, and for any type of access.

An XFI module has access only to a certain set of contiguous memory regions; for each type of access, a certain range [A,B] may be special, as explained below. For accesses to constant addresses and to the scoped stack, this property is established by static verification. Guards check other accesses at runtime.

XFI memory-range guards ensure that a register holds an accessible address; moreover, they ensure that a range around this address is also accessible (within non-negative constant offsets L, below, and H, above). Control-flow integrity implies that a single such guard can protect multiple memory accesses—namely, those that it dominates, use the same register value, and remain within the range determined by L and H. Similar (but more complex) memory-range checks ensure that only accessible addresses are used in simple loops such as the x86 REP instructions. In this case, L and H may be variable and reside in registers.

The comparisons in memory-range guards must take into account the sizes of memory accesses, and H must be chosen accordingly. For example, the guard for a

```
# mrguard(EAX, L, H) ::=
    if EAX < A + L, goto S
    if B - H < EAX, goto S
    Mem[EAX] := 42
M:
                        # Two writes
    Mem[EAX - L] := 7
                        # both allowed
S:
    push EAX
                        # Arguments for
    push L. H
                        # slower guard
    call SlowpathGuard
    jump M
                        # Allow writes
```

Figure 3: Two memory writes, and a memory-range guard for the range [EAX-L, EAX+H). The guard executes faster if this range lies within [A, B). The constant H should be at least 4.

four-byte write through EAX must ensure that all bytes in the range [EAX,EAX+4] are writable. The guards must also handle corner cases, such as when A is near the end of addressable memory. In the common situation where A+L and B-H are constants established by the loader (as described in Section 4.1), the corner cases can be treated statically. For variable memory ranges, like those of x86 REP loops, guards must perform additional runtime checks. The guards must also consider the possibility of arithmetic overflows.

Figure 3 shows an example memory-range guard that establishes that access is allowed to all addresses in the range [EAX-L,EAX+H). The figure includes two write instructions that each accesses four bytes of memory through EAX. For the guard to check the writes correctly, the offset H must therefore be at least 4. The figure also introduces a shorthand mrguard. Below, mrguard (EAX, L, H) represents the contents of the figure minus the two writes. As shown in the figure, memory-range guards can be implemented as two paths: one faster, the other slower.

- The fastpath directly compares an address in a register with the values A+L and B-H. The guard permits access if the address lies within these bounds.
- If the fastpath comparisons fail, then the guard calls a host-system slowpath with appropriate arguments. The slowpath searches to see if the address range lies within any other accessible memory regions. The address, and the values L and H, are parameters to this search. The search itself may be arbitrary code, and may for example involve direct comparisons or data structures similar to page tables. The search may be invoked by a direct jump, a trap or fault (e.g., the x86 bounds exception), or by other explicit or implicit control flow.

Analogously, CFI guards could also use a slowpath when embedded identifiers do not match, thus supporting a rich, dynamic notion of validity for function pointers. We have not yet needed this extension.

4 Implementation

We have designed and built an XFI implementation. This section describes it, and also considers optional, simple, specialized hardware support for XFI. Although still a prototype, the implementation is already complete enough to be practical, as demonstrated by the applications and measurements described in later sections.

4.1 XFI Modules

XFI modules are dynamically loadable executables in an appropriate object format. In our implementation, they are Windows "Portable Executable" binaries, often named EXEs or DLLs [34].

Modules consist of multiple sections, which may have different access permissions. Machine code is in one executable section, and program data, such as read-only constants and writable global variables, is in others. Data in import and export sections allow the determination of module entry points and use of host-system support routines. Other sections, and the module header, provide host systems with auxiliary data (for example, cryptographic signatures and offsets for load-time relocation). Auxiliary sections are used only at load time.

XFI protection thus relies on several module sections. Import sections, as well as host-system policy, limit an XFI module's use of support routines; similarly, memory access is constrained by section access permissions. The relocation section may give values to XFI constants, such as the constants A+L used in memory-range guards. Finally, a new, auxiliary section holds untrusted verification hints (discussed in Section 4.3).

The efficiency of memory protection can benefit from choices in the structure of XFI modules. Specifically, writable fastpath memory may lie completely within a read/write section of each XFI module: once a module is loaded, this section can have any amount of memory. (A section-header value gives its size.) In this case, the module, or its host system, may provide a heap implementation that allocates memory within that section. Such a fastpath region can be made large without wasting physical memory: host systems with virtual memory support can allocate physical pages as requested by the XFI module (e.g., by the heap implementation calling a support routine). This strategy is especially attractive for 64-bit systems, where ample virtual size can be given to a fastpath region.

4.2 The Rewriter

We produce XFI binary modules from Windows x86 executables with a rewriter based on the Vulcan library [37]; XFI rewriters could as easily be created using similar libraries for other architectures [37, Section 7]. Although our x86 rewriter requires neither recompilation nor source-code access, it makes use of debug informa-

80

tion (PDB files), for instance to distinguish code from data. Such PDB files are publicly available for all Windows components. Our rewriter is relatively complex and not very fast. In particular, it must do intra-procedural analyses for control flow, stack use, and register use; these analyses are not always linear.

Our rewriter does not currently handle certain hand-written modules; it also stumbles on frame-pointer-optimized code, and code with certain variable-size stack allocation. (Both can be supported using a virtual register.) Similarly, our verifier does not handle MMX, SSE, and other x86 instruction-set extensions. However, since our verifier is conservative, neither these limitations nor rewriter bugs should ever result in the execution of an XFI module without XFI protection.

Alternatively, XFI modules could be output by the code-generation phase of a compiler. A compiler that would produce XFI modules would generate mostly standard machine code, but add structured inline guards and verification hints, and use the allocation stack for stack data subject to computed memory access. Such a compiler could make more thorough use of XFI mechanisms, help reduce overhead, and remove some limitations of our prototype.

4.3 Verification

As discussed in Section 3, the correctness of XFI protection depends on the load-time verification of XFI modules. Our verifier was written from scratch and is self-contained. In particular, it is independent from our rewriter, and from any specific strategy for creating XFI modules. It is 3000 lines of straightforward, commented C++ code, most of which are tables for x86 opcode decoding. The verifier needs only a basic understanding of x86 behavior, modeling nothing more complex than integer comparisons and how instructions copy registers. Its simplicity contributes to our confidence in the verifier.

The verifier is much simpler than the rewriter because the verifier does only local reasoning about individual basic blocks. This reasoning takes advantage of a set of untrusted verification hints that must be present in the XFI module. A similar strategy is used by Java bytecode verifiers and in PCC systems.

Therefore, the verifier is not only simple but also fast. It makes a linear pass over the bytes of an XFI module, doing mostly instruction decoding and comparisons.

In order to establish the correct use of other XFI machinery, the verifier checks several specific conditions. These conditions refine and strengthen the requirements outlined in Section 3. In particular, the verifier ensures that both the allocation stack and the scoped stack are managed properly: the allocation-stack and scoped-stack pointers are updated only by bounded, constant amounts, the code contains stack overflow guards, the return ad-

dress and other virtual registers are saved and retrieved from the scoped stack only, and function calls preserve the heights of both stacks.

Verification proceeds by considering the execution of machine-code instructions abstractly; it manipulates verification states which are predicates that describe concrete execution states. In particular, verification states model register contents, including the contents of the scoped stack which holds function-local virtual registers. The execution of each instruction $\mathbf i$ can be represented by a Hoare triple: $\{P\}$ $\mathbf i$ $\{Q\}$, where P and Q are verification states. Given P, the verifier ensures that P guarantees the safe execution of $\mathbf i$; the verifier also computes Q and ensures that Q implies the verification states before each possible successor instruction.

To simplify verification, XFI modules must include a set of verification hints that guide the verifier. For the most basic version of the verifier, the hints must provide the verification state for the entry to each basic block. The verification is done one basic block at a time, and on basic-block exit the verifier checks that the final, computed verification state implies the verification state at entry to all possible successor blocks.

As mentioned above, the verifier also ensures that the proper CFI guards precede computed control-flow transfers. The verifier expects hints to specify the set of possible targets of computed jumps; these hints allow the verifier to collect the set of CFI target identifiers used in the module. The verifier scans the machine code of the module in order to ensure that these identifiers occur only at the beginning of basic blocks. Finally, the verifier allows computed control flow only when the verification state records the effects of an appropriate CFI guard. Similarly, the verifier allows a computed memory access only when the verification state records the effects of an appropriate memory-range guard.

Figure 4 shows an example program fragment. The memory-range guard in line 0 checks that the memory in the range [EAX-0,EAX+8) lies within a single accessible memory region. (See Figure 3 for a definition of mrguard.) This fragment copies the word stored at address EAX to address EAX+4, loads the result value from the allocation stack into EAX (in line 3), then restores the allocation-stack pointer (ASP) by loading it from a virtual register on the scoped stack, and returns. As shown in the comments, the pop and return instructions use the scoped stack.

At the start of verification of the code in Figure 4, the verification state encodes that the original value of the scoped stack pointer is 8 more than the current value, the return address is in the virtual register at address SSP+4, the value on function entry of the allocation-stack pointer is stored in the virtual register at address SSP, and the allocation-stack frame range [ASP-32, ASP) falls into a

```
{origSSP=SSP+8, valid[SSP, SSP+8) }
{retaddr=Mem[SSP+4]}
{origASP=Mem[SSP], valid[ASP-32, ASP)}

0. mrguard(EAX, 0, 8)
{valid[EAX-0, EAX+8)}

1. EDX := Mem[EAX]

2. Mem[EAX + 4] := EDX

3. EAX := Mem[ASP - 4]

4. pop ASP # ASP := Mem[SSP]; SSP := SSP+4
{origASP=ASP, valid[SSP, SSP+4)}
{origSSP=SSP+4, retaddr=Mem[SSP]}

5. ret # SSP := SSP+4; jump Mem[SSP-4]
```

Figure 4: Verification states for a small fragment of XFI module code. The items written in braces correspond to the verification state at the given program point.

contiguous accessible region of memory. For clarity, Figure 4 shows only modifications to the verification state. The verifier recognizes the meaning of the instructions of the memory-range guard in line 0, and adds the corresponding "valid" fact to the verification state for the following instruction. All of the previously known facts are preserved in this case, and the resulting "valid" facts suffice for checking all the memory accesses in the program fragment. In line 4, the verification state is changed to reflect the arithmetic operation implicit in the pop instruction. After line 4, the verification state considers valid only the range [SSP, SSP+4), since in systems code the contents of the stack below SSP can always be clobbered by interrupts. At line 5, the verifier checks that the stack pointers will be restored to their original values before the return, and that the proper return address is used. In order to enable this reasoning, the verifier includes support for manipulating linear equalities and inequalities.

4.4 Inline Guards

Many considerations influence the choice of the exact machine-code sequences for XFI mechanisms such as the inline guards. Finding suitable, efficient guards can be particularly challenging for the x86, because of its complex, non-uniform instruction set, and dearth of registers. In our XFI implementation, we address these x86 kinks partly through a register-liveness analysis that the rewriter employs to discover or make available free registers or x86 condition flags. We also use the following specific code sequences for each type of guard.

CFI Guards Our x86 CFI guards are implemented in much the same manner as shown in Figure 2, and detailed in [2]. Instead of having identifiers precede valid destinations, we embed callee identifiers within an instruction, prefetchnta, that has few side effects.

```
EAX := SSP - L - K
if EAX < Mem[FS + 8], goto SSPERR
C: SSP := SSP - L # Lower SSP by L</pre>
```

Figure 5: Scoped-stack overflow guard on Windows.

Stack-Overflow Guards We employ memory-range guards in order to prevent allocation-stack overflow: if ASP is to be lowered by L, the range [ASP-L,ASP) must lie within an accessible region.

Our guard against scoped-stack overflow exploits a particular property of x86 Windows: the bottom of the current stack is held in directly accessible, thread-local storage pointed to by the FS segment register. (Future, Windows-specific implementations might usefully keep XFI-specific data in this thread-local storage.)

Figure 5 shows an x86 scoped-stack guard; it compares a proposed SSP value with the stack limit at FS+8 in the thread control block. Our guards ensure that free space always remains at the bottom of the scoped stack (e.g., for interrupt state). Thus, in the figure, code at label C lowers SSP by L, but our overflow guard ensures that K more space is available, where K is a constant.

Allocation-stack overflow guards can also be implemented in a host-specific manner. For instance, for single-page allocation stacks (such as may be used in the Windows kernel), an invariant on the top bits of ASP could allow for a fast guard—even when allocation stacks are placed in slowpath memory.

Memory-Range Guards Our implementation uses a fastpath memory region whose endpoints A and B are embedded in the memory-range guards. This region lies within the XFI module, as suggested in Section 4.1; its endpoints are relocation constants set during loading. In theory, the endpoints could be fairly arbitrary, even values held in reserved registers—although our implementation does not spend x86 registers for this purpose.

Our x86 memory-range guard is like that shown in Figure 3 of Section 3. The slowpath call is not placed inline, both because it would waste code cache and also because the x86 predicts forward branches as not taken, by default. We have considered having our guards raise exceptions, caught by a slowpath guard provided by the host system; however, this approach is likely to be less efficient than guards that call slowpaths. In particular, our implementation cannot use the x86 BOUND instruction because hardware bounds exceptions are treated as fatal conditions in the Windows kernel.

4.5 Host-System Support

Most XFI modules need runtime support from their host system, as discussed above. This section describes the support components that host systems must implement in order to make use of XFI protection.

As an alternative, instead of the host system implementing them separately, these support components may be included with XFI modules, for instance in a distinct library-code section. Such library code may be trusted because of cryptographic signatures or for other reasons; therefore, this code may even include the XFI verifier.

Slowpath Permission Tables When executing their slowpath, memory-range guards must search permission tables: runtime data structures, maintained by the host system, that hold the set of accessible memory regions. There may be multiple such tables, one for each type of slowpath access. Each search checks whether a range [R-L,R+H) lies within a contiguous memory region to which the host system has granted access.

Searching these tables must be fast, as some software will access memory in a manner that frequently calls the slowpath of memory-range guards. Fortunately, fast permission tables can be implemented in several ways. In particular, by using tables similar to page tables, searches can use known, efficient techniques for software-filled translation lookaside buffers, or TLBs [43].

In our implementation, permission tables are very simple: a null-terminated list of address pairs, of the start and end of regions. Even though there are faster alternatives, we chose this representation not only because it is simple to maintain and search, but also because our experiments indicated that more complex tables would be of limited benefit.

Allocation-Stack Manager A host system must properly associate an allocation stack with each thread that executes in an XFI module. Allocation-stack management must consider both performance and resource consumption. For instance, a thread can keep using the same allocation stack if it calls an XFI module reentrantly; alternatively, it may adopt a new allocation stack on each entry.

Our implementation uses a pool from which hostsystem threads draw allocation stacks when they call XFI modules. The size of the pool is adjusted on the basis of the concurrency in the XFI module. The pool's data structure is a simple array, guarded by a single lock; this array is consulted in order to acquire and release allocation stacks, as threads go through software call gates.

Software Call Gates An important component of XFI is the software that mediates calls to entry points and support routines, in particular to map to and from the two-stack model used by XFI module execution. These wrappers are a software form of call gates, as found in some hardware architectures [43]. They will typically be implemented by the host system, and may perform all the work necessary to manage stacks and to maintain permission tables. Our wrappers, in addition, resolve complications such as the multiple calling conventions of x86

Windows and variable-argument support routines.

Instead of copying arguments and results, as in component systems [7], our wrappers can edit the slowpath permission tables to marshal access rights on calls and returns. This alternative is important when it is not desirable to copy into and out of XFI module fastpath memory, such as for large, infrequently used data buffers. Device drivers may be good candidates for this optimization, as they often manage large buffers, while performing little processing of those buffers. (For example, a storage driver may never examine the data being read or written.)

Windows Exception Dispatcher The Windows software execution model includes a unified mechanism for handling software and hardware exceptions, both synchronous and asynchronous. This Structured Exception Handling (SEH) [31, 34] applies both in user mode and in the kernel; it generalizes Unix signals and C++ exceptions, and can be used to implement both. On the x86, SEH requires functions that catch exceptions to place relevant SEH metadata on the stack, including the address of a catch expression and a catch handler. (On the other hand, 64-bit Windows places the metadata for all functions in a static, read-only table.) Function prologues and epilogues link this metadata into a chain of handlers, whose head is at FS+0 in the thread control block. Upon an exception, catch expressions along this chain are evaluated, determining which catch handler is used. Multiple, strong invariants apply to this metadata, as well as to the handler chain.

XFI protection for Windows must consider SEH, both to enforce policies (e.g., P6), and to support correct execution of Windows software, which often makes use of SEH. In particular, XFI should allow both catch expressions and catch handlers to run on both XFI stacks, and (possibly) allow control to resume at the faulting instruction. Fortunately, XFI mechanisms such as virtual registers enable SEH integrity to be fully maintained. This integrity results in important security benefits: SEH corruption is a favorite means of attack on Windows [32].

We have designed support for SEH in our XFI implementation; we outline it next (omitting details because of space constraints). We have not yet fully built this support, but our implementation does handle Unix-style signals. The support includes both the necessary host-system components and what conditions are checked by the XFI verifier. All exceptions are directed to our exception-dispatcher implementation in the host system; it saves registers, and invokes the XFI module's SEH code in the proper order, and using the right context (e.g., for the ASP register). The XFI verifier ensures each function's SEH metadata is well formed, and used appropriately; it also restricts catch expressions so they properly access stacks via a base pointer.

4.6 Possible Architecture Support

The specifics of XFI mechanisms and their behavior depend on hardware characteristics. In particular, the code and the cost of software guards will vary across platforms. Therefore, we have designed and evaluated relatively straightforward hardware support for CFI guards and memory-range guards in an Alpha simulator. (No precise x86 simulator is available.) This support extends the Alpha instruction set with a few, new register-based comparisons instructions.

We implemented CFI guards with four new instructions: a cfilabel instruction and a variant of each of the Alpha computed transfer instructions ("indirect jump", "return", and "jump to subroutine"). The instructions contain 16-bit, immediate identifiers. After a CFI transfer instruction with identifier ID, a cfilabel with ID must be executed before any other type of instruction; otherwise, a hardware exception is triggered. Between these instructions, the value ID is stored in a normal, renamed register, so multiple CFI guards can be in flight simultaneously. To avoid reads of this CFI register at each instruction, it is monitored with a two-state automaton in the commit stage of the pipeline. An x86 implementation could be similar, except that it might use instructions up to six bytes long (e.g., in order to allow larger identifiers) and an explicit CFI register.

We implemented memory-range guards with three new instructions, one for each type of access (read, write, and execute). Each instruction takes the form mrguard R, L, H, naming a register R, and providing L and H as 10-bit immediate constants. The instructions compute values R-L and R+H and compare them with registers that hold A and B, using the regular, scheduled arithmetic units of the processor. (Thus, the instructions implement mrguard as defined in Figure 3.) On fastpath failure, our instructions (like the x86 BOUND instruction) throw a hardware exception that provides R, L, and H only as implicit arguments via the faulting instruction. An x86 mrguard instruction could perform a real call, and push arguments onto the stack, with accordingly higher performance; it could be implemented in microcode, with the address of the slowpath function to call in a machinespecific register.

We studied these new instructions for XFI guards in a validated, cycle-accurate Alpha EV6 simulator [14]; the details are in a companion report [9]. Our results indicate that hardware-supported XFI can be quite efficient: adding effect-free NOP instructions instead of XFI guards leads to virtually identical overhead as using the new instructions. In part, these results reflect that our XFI instructions introduce no new dependencies, and can leverage unused processor units; they also reflect cache pressure due to increased code size. These results are likely to carry over to the x86 architecture. Section 6.2

gives measurements of the overhead of using x86 NOPs in place of XFI guards.

5 Applications

XFI has a wide range of potential applications. To date, we have applied our XFI implementation to dynamic libraries, device drivers, and multimedia codecs. In this section we describe some of these applications; for brevity, we focus on device drivers.

Device drivers are a prime application domain for XFI protection, both because they should comply with XFI policy and also because their failure to do so has serious consequences.

We have implemented XFI for kernel-mode device drivers developed using the Windows Driver Foundation (WDF)—a new-generation framework, designed to simplify the development of correct drivers for all versions of Windows [26]. With WDF, a device driver no longer has direct access to kernel abstractions, such as work-item queues, request packets, or device objects; rather, these are opaque to the driver, and WDF performs all modification and synchronization on the driver's behalf. In addition to holding abstract such critical state, WDF validates that drivers comply with contracts, even on production systems. Thus, WDF represents a significant advance in interface safety over previous driver support frameworks [30].

XFI protection can isolate device drivers in a WDF host system. Without this isolation, despite all its interface-safety checks, it is impossible for WDF to contain driver faults (or exploits) fully, or even to detect them in all cases. Since WDF mediates on all interactions with the kernel, device drivers interact with their host system frequently; therefore, XFI protection is more attractive than potential alternatives with higher context-switch latency.

We have added to WDF the necessary components for XFI protection. These include the machinery described in Section 4, as well as some WDF-specific machinery. In particular, this machinery enforces the assured self-authentication policy (described in Section 3), as follows. Since WDF can serve multiple drivers simultaneously, drivers pass along a pointer to their WDF control block when they call support routines. A driver may attempt to spoof this value in order to call WDF as another driver. We thwart such impersonations by requiring the first argument to each WDF support routine to be a specific, read-only variable in the XFI module of the calling driver. WDF has write access and sets this variable as it loads the driver.

It is simple to determine the set of memory regions that a WDF driver should be allowed to access. A driver gains access to memory either through explicit allocation—a call to a WDF support routine—or implicitly, such as when an I/O buffer is passed to the driver for processing. In all cases, WDF has the information necessary to grant (and later revoke) access to those memory regions, by modifying the slowpath permission tables.

We have applied XFI to some WDF drivers, including a RAM disk driver and a benchmark driver used by the WDF team to measure performance. Section 6 gives the results of these experiments.

Multimedia codecs are another attractive application domain for XFI protection. They are typically extensions (often downloaded, untrusted code) that operate on data at the request of their host system. Furthermore, they have been the subject of many successful attacks. As described in Section 2, we have turned the standard JPEG implementation of image decoding [20] into an XFI module. Section 6 gives the results of these experiments as well.

6 Evaluation

We have performed a number of experiments and analyses in order to evaluate the benefits and overheads of XFI protection. This section summarizes our results.

6.1 Protection Benefits

XFI protection against faults such as spurious writes to memory is similar to the protection of hardware-supported address spaces (e.g., processes). However, some aspects of XFI protection do not lend themselves to simple comparisons; for instance, XFI enforces integrity guarantees that protect against security exploits. We have validated these XFI benefits, in part by experimenting with actual exploits.

We have established that CFI guards alone are sufficient to prevent a wide variety of attacks that follow deviant machine-code execution paths and thereby violate XFI policy. In particular, CFI blocks the exploits used by Blaster and Slammer and those in a test suite of 18 other attack vectors; CFI also thwarts the published exploits of a heap overflow in the widespread software used in our JPEG module. More details can be found in [2].

XFI also helps defend against data-corruption exploits For example, in an XFI module, if a global configuration variable does not reside in a region that is subject to computed memory access, then that variable can be modified only by those instructions that refer to it by name; similarly, access to function-local variables is limited. In this manner, XFI program-data integrity can effectively thwart exploits described in a recent paper on data-only attacks [12].

Unfortunately, some attacks still succeed, despite XFI protection. Most notably, these include Nimda-like attacks that abuse over-permissive support routines. Defense against such attacks remains an open problem.

6.2 Enforcement Overhead

We applied our x86 XFI implementation to WDF device drivers, SFI benchmarks [36], the JPEG decoder [20], and Mediabench kernels [22], all compiled using Microsoft VC++ 8.0, with optimizations. Rewriting each module took a few seconds; verification is linear in the module size and typically takes a few milliseconds. We made elapsed-time measurements for a Pentium M 1.5GHz processor, on an idle system with daemon services disabled. The processor was fully utilized in all runs. (For drivers, 90% of the time was in the kernel.)

Tables 1, 2, and 3 summarize the results of our measurements; the tables report overheads relative to modules before XFI rewriting. Overheads are the average of five runs of more than ten seconds, with standard deviation less than 1.5%. Overheads are shown for *write protection*, which restricts only writes, and (between parentheses) for *read-write protection*, which also restricts reads. We emphasize write protection because integrity (rather than confidentiality) is usually the central goal of protection.

Our tables have three columns of results. The slowpath column shows overhead when memory-range guards have to consult permission tables for access to the data processed by the XFI module. The fastpath column shows results when data already resides within the module's fastpath region—for instance, because the host system passed a copy of the data when it called the module entry point. Both columns reflect the case where XFI modules have a private, fastpath memory. Memory-tomemory copies are fast on modern processors, and fastpath regions can be large (as discussed in Section 4.1), so host systems may find it most convenient to copy arguments to and from XFI modules; in this case, only fastpath overheads apply. Slowpath memory, and the techniques of Section 4.5, may be used only rarely, in particular when state truly needs to be shared.

The NOP column shows the measured overhead when each inline guard is replaced with a six-byte x86 NOP instruction. Therefore, it gives a baseline for the cost of structuring binaries as XFI modules, and of including guards in the appropriate places. In particular, the NOP column shows overheads that results from increased cache pressure and from additional instruction decoding. As discussed in Section 4.6, NOP overheads may also be indicative of the overheads that would be seen with hardware-supported XFI: on the x86, six bytes are a reasonable size for new XFI guard instructions.

Each table also shows the code size of each XFI module as a multiple of the code size of the original binary. The size increase results from inline guards, and is often substantial; however, the added code is often not executed, since our x86 guards are implemented with an out-of-band slowpath (as shown in Figure 3).

		NOP	fastpath	slowpath
hotlist	Δ sz	2.1x (2.6x)	2.5x (4.1x)	3.9x (8.3x)
nothst	%	1% (5%)	4% (94%)	5% (798%)
lld	Δ sz	1.2x (1.3x)	1.5x (1.8x)	1.7x (2.3x)
	%	10% (28%)	27% (60%)	93% (346%)
MD5	Δ sz	1.1x (1.1x)	1.2x (1.3x)	1.3x (1.5x)
	%	-1% (2%)	3% (7%)	27% (101%)

Table 1: Code-size increase and slowdown of SFI benchmarks, with XFI. The % rows show slowdown.

	Kt/s	NOP	fastpath	slowpath
Δ sz		1.3x (1.3x)	1.3x (1.4x)	1.4x (1.6x)
1	193	5.0% (4.8%)	6.8% (6.1%)	5.9% (13.4%)
512	151	4.7% (3.9%)	5.3% (4.7%)	4.8% (10.6%)
4K	71	1.7% (1.7%)	2.7% (2.9%)	2.6% (5.0%)
64K	5	1.2% (1.9%)	1.4% (0.4%)	1.7% (1.8%)

Table 2: Code-size increase and slowdown for different kernel buffer sizes for a WDF benchmark, with XFI. The unprotected driver is 11KB of x86 machine code; its transactions per second (shown in thousands) form the baseline for the slowdown percentages.

Table 1 shows numbers for well-established benchmarks for SFI performance [16, 25, 36]; each is less than 3K of machine code. Unlike XFI, SFI supports only one accessible memory region, so only XFI fastpath overhead is directly comparable with published SFI measurements. This overhead is either similar to that of SFI or significantly lower. For instance, XFI fastpath overhead on MD5 is only 3%; for SFI, the corresponding numbers range from 23% [36] to 47% [25]. XFI can have lower overhead than SFI because, with XFI, a single memory-range guard can suffice for multiple memoryaccess instructions. The hotlist benchmark searches a linked list, in a tight loop; each pointer must be checked separately. Therefore, its read-write slowdown (up to nine-fold) reflects the use of a memory-range guard every few instructions.

Table 2 gives the results of running a WDF benchmark with XFI protection, using KMDF 1.0 [26]. In the benchmark, a user-mode program stores in a kernel-mode driver a data buffer of a certain size; this buffer is then retrieved and validated. Table 2 exhibits clear, understandable trends: enforcement overhead is reduced by using either a larger buffer per transaction, or faster and fewer guards. (The table also exhibits small anomalies: sometimes the use of more, slower guards slightly improves performance, possibly because of cache effects.) For this benchmark it was important that XFI is able to guard use of the x86 REP instructions; otherwise, a guard might have to be executed for every buffer byte, resulting in up to three-fold slowpath overhead.

In addition, we experimented with a WDF implementation of a RAM disk driver. We copied 8 megabytes

	NOP	fastpath	slowpath	
Δ sz	1.3x (1.6x)	1.7x (2.5x)	2.1x (3.7x)	
4K	14% (34%)	18% (78%)	42% (112%)	
14K	15% (36%)	18% (80%)	43% (116%)	
63K	12% (31%)	17% (75%)	40% (108%)	
229K	11% (28%)	15% (68%)	35% (98%)	

Table 3: Code-size increase and slowdown for different-size input data for JPEG decoding, with XFI. The unprotected decoder is 59KB of x86 machine code; the baseline for the slowdown shown is decoding time.

	NOP	fastpath	slowpath
adpcm_encode	0% (4%)	2% (49%)	13% (149%)
adpcm_decode	-3% (2%)	3% (12%)	36% (112%)
gsm_decode	3% (1%)	79% (97%)	125% (230%)
epic_decode	3% (9%)	7% (19%)	119% (220%)

Table 4: Slowdown of Mediabench kernels, with XFI.

back and forth, between two directories on the RAM disk, flushing the file cache after each copy. We repeated these copies several thousand times, both using single-byte files and files of varying sizes. The RAM disk did not exhibit any measurable end-to-end slowdown, even with slowpath guards. (Similarly, for many modules, the overhead of XFI protection may be overshadowed by other processing.)

Between the above drivers and the WDF, we measured up to 900,000 transitions per second—but each had little cost. In comparison, placing drivers in hardware-supported address spaces causes overhead on each transition (because new page tables must be loaded), as well as during code execution (because of higher TLB miss rate). Such overhead is correlated with the rate of transitions between the kernel and the driver, and at 23,000 transitions per second it can cause more than three-fold slowdown of kernel processing, as reported in [39]. Our experiments concern WDF drivers that have a much higher transition rate; correspondingly, their processing slowdown with hardware-based protection could be much worse. With XFI protection, these hardware-related overheads are fully eliminated.

Table 3 shows results for the JPEG decoder for inputs of four different sizes. Each of the inputs yields an image about eight times larger. Although JPEG processes images in 64-pixel blocks, overhead is somewhat smaller for larger images—again because of fast guards for REP loops. Table 4 shows results for other multimedia codecs, namely three Mediabench kernels (defined as in [9]). Overall, the performance of these XFI multimedia codecs seems acceptable, considering that they are based on unmodified, standard C-language sources, and that they can be used safely within any x86 hardware protection ring.

7 Related Work

Since there is a wealth of work on protection, we briefly review the landscape and include details on only a few specific pieces of related research.

Unfortunately, no single approach to protection will be suitable for all scenarios in modern, commodity systems. Some approaches, like Nooks, are designed for "fault resistance, not fault tolerance" and can be easily circumvented by malicious code [39]. Others, like Mondrix, require sophisticated, new hardware support, or changes to the system foundations for the inclusion of a protection supervisor [4, 23, 44]. In its published incarnations, SFI [16, 25, 36, 41] has enforced a policy more basic than that of hardware-supported address spaces, yet placed hard-to-meet constraints on memory layout and other system aspects, such as signals and multithreading. Language-based approaches [5, 19, 24, 28] usually require re-writing software; they can also require elaborate runtime support that may be difficult to include at all levels of a system.

XFI is founded on the view that protection should be mostly a software issue [6] (cf. [42]). In comparison with traditional hardware-supported address spaces, software machinery can offer almost arbitrary expressiveness. It may also be easier to deploy than new architectures (e.g., [44]). Finally, it can make it efficient to switch protection domains, and it avoids technical difficulties such as the efficient hardware implementation of permission tables. Of course, even software-based approaches require appropriate hardware support [6], sometimes to a non-trivial extent (e.g., x86 segments for protecting a call stack in our previous work on SMAC [2]).

Software protection systems vary widely in their contexts, goals, and techniques:

- Some systems address protection between kernel environments (e.g., [4, 23, 39]), while others target applications isolated by typing (e.g., [19]). XFI aims to be applicable in a broad range of contexts.
- Inlined reference monitors [17] and PCC [29] aim
 to support the enforcement of a large class of properties. More targeted systems include ours for CFI.
 XFI leverages CFI in order to offer, systematically,
 external properties and critical-state integrity. Going beyond protection, some software techniques
 also address rollback and recovery (e.g., [38]); we
 have not yet studied them in the context of XFI.
- Some systems rely on interpretation techniques; program shepherding [8,21] and the use of virtual machine monitors [4,23,44] are examples of this approach. Many others (including XFI) rely on language constraints, static and dynamic typing, and other kinds of analyses [15, 19, 27, 33]. Generally, static analysis aims to verify that dynamic checks

are not necessary. In XFI and a few other recent systems with other objectives (e.g., [2, 10, 18]), the static analysis aims to ensure that dynamic checks have been applied properly, and it is the responsibility of an independent verifier.

XFI is most closely related to SMAC, mentioned above, and to SFI implementations such as MiSFIT [36]. Indeed, one of the motivations for XFI was our desire for a practical protection mechanism in the spirit of SFI. Originally, SFI was a carefully designed system for enforcing a relatively basic protection policy, with impressive performance [41]. Its RISC-based implementation relied on several significant assumptions: fixed-length instructions, several registers free for dedicated use, a single aligned, contiguous block of memory, and system support based on hardware protection. While these assumptions can be easily satisfied in a new, 64-bit RISC system, they are problematic in many other settings such as x86 commodity systems. Later SFI implementations have added further assumptions about memory use and alignment, trusted compilers, and hardware support [16, 25, 36]; some maintain additional invariants on certain state, in a limited fashion (e.g., on return addresses in single-threaded programs [36]). In particular PittSFIeld [25] is an attractive, new SFI implementation that applies to CISC architectures. PittSFIeld achieves efficiency by using hardware support for guard pages, reserving large, aligned regions of memory, and not handling race conditions on function returns. Furthermore, it offers limited protection for kernel-mode code (e.g., as it pops the x86 flags register from arbitrary memory).

In comparison with SFI, XFI supports richer policies, for instance for fine-grained access control and the limited, safe use of privileged instructions. Because it depends on few assumptions, XFI is applicable to legacy software, on commodity systems, and even despite vulnerabilities such as buffer overflows. While some SFI performance improvements reduce fault isolation, XFI guarantees protection even with optimizations. Finally, XFI does well in terms of performance; for example, our basic XFI implementation has lower overhead than PittS-FIeld for JPEG, and it runs MD5 faster, even when inputs are in slowpath memory.

8 Conclusions

XFI protection allows code to be executed with strong safety and security guarantees, without the need for a separate process or the creation of new software written in a type-safe language. Like hardware protection, however, XFI addresses low-level architectural features; like language-based protection, it leverages static analysis and offers expressiveness. Thus, XFI inhabits an interesting and practical middle ground. In this respect, XFI has much in common with SFI and PCC. Building

on SFI, PCC, and related efforts, the design and implementation of XFI include a number of ideas and techniques that contribute to its flexibility, completeness, efficiency, and wide applicability. As a result, XFI offers protection even for legacy code that is run natively in the most privileged ring of x86 systems.

Acknowledgments This work was done at Microsoft Research, Silicon Valley. John Richardson motivated and supported our WDF work. Venugopalan Ramasubramanian helped with experiments. We are grateful to Brad Chen for his careful shepherding, and to the OSDI reviewers and Greg Morrisett for their useful comments.

References

- M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. A theory of secure control flow. In *International Conf. on Formal Engineering Methods*, 2005.
- [2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In ACM Computer and Communications Security Conf., 2005.
- [3] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In POPL, 2000.
- [4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In SOSP, 2003.
- [5] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In SOSP, 1995.
- [6] B. N. Bershad, S. Savage, P. Pardyak, D. Becker, M. Fiuczynski, and E. G. Sirer. Protection is a software issue. In *HotOS*, 1995.
- [7] D. Box. Essential COM. Addison-Wesley, 1997.
- [8] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In Symp. on Code Generation and Optimization, 2003.
- [9] M. Budiu, Ú. Erlingsson, and M. Abadi. Architectural support for software-based protection. Technical Report MSR-TR-2006-115, Microsoft Research, 2006.
- [10] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *European Symp. on Programming*, 2005.
- [11] B.-Y. E. Chang, A. Chlipala, and G. C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *International Conf. on Verification, Model Checking,* and Abstract Interpretation, 2006.
- [12] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. Iyer. Non-controldata attacks are realistic threats. In *Usenix Security*, 2005.
- [13] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security*, 1998.
- [14] R. Desikan, D. C. Burger, S. W. Keckler, and T. Austin. Simalpha: a validated, execution-driven Alpha 21264 simulator. Technical Report TR-01-23, U.T. Austin, Dept. of C.S., 2003.
- [15] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *Information Processing (Proc. of the IFIP Congress)*, 1971.
- [16] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In New Security Paradigms, 1999.
- [17] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symp. on Security and Privacy*, 2000.
- [18] K. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In ACM Workshop on Program-

- ming Languages and Analysis for Security, 2006.
- [19] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.
- [20] Independent JPEG Group. Reference implementation for JPEG image compression, 1998. http://www.ijg.org/.
- [21] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Usenix Security*, 2002.
- [22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In MICRO, 1997.
- [23] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In OSDI, 2004.
- [24] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, 1996.
- [25] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Usenix Security*, 2006.
- [26] Microsoft Corp. Windows Driver Foundation (WDF), 2006. http://www.microsoft.com/whdc/driver/wdf/.
- [27] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. ACM Operating Systems Review, SIGOPS, 21(5), 1987.
- [28] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In POPL, 1998.
- [29] G. C. Necula. Proof-carrying code. In POPL, 1997.
- [30] W. Oney. Programming the Microsoft Windows Driver Model. Microsoft Press, second edition, 2002.
- [31] M. Pietrek. A crash course on the depths of Win32 structured exception handling. *Microsoft Systems Journal*, 1997. http://www.microsoft.com/msj/0197/.
- [32] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4), 2004.
- [33] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Ottawa Linux Symp.*, 2005.
- [34] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals*. Microsoft Press, fourth edition, 2004.
- [35] S. Savage and B. N. Bershad. Some issues in the design of extensible operating systems. In OSDI, 1994.
- [36] C. Small and M. I. Seltzer. MiSFIT: Constructing safe extensible systems. *IEEE Concurrency: Parallel, Distributed and Mobile* Computing, 6(3), 1998.
- [37] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.
- [38] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In OSDI, 2004.
- [39] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In SOSP, 2003.
- [40] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for Internet services. In SOSP, 2003.
- [41] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In SOSP, 1993.
- [42] E. Witchel and K. Asanovic. Hardware works, software doesn't: Enforcing modularity with Mondriaan memory protection. In *HotOS*, 2003.
- [43] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In ASPLOS, 2002.
- [44] E. Witchel, J. Rhee, and K. Asanović. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In SOSP, 2005.

Operating System Profiling via Latency Analysis

Nikolai Joukov¹, Avishay Traeger¹, Rakesh Iyer¹, Charles P. Wright^{1,2}, and Erez Zadok¹

¹Stony Brook University

²IBM T. J. Watson Research Center, Hawthorne

Abstract

Operating systems are complex and their behavior depends on many factors. Source code, if available, does not directly help one to understand the OS's behavior, as the behavior depends on actual workloads and external inputs. Runtime profiling is a key technique to prove new concepts, debug problems, and optimize performance. Unfortunately, existing profiling methods are lacking in important areas—they do not provide enough information about the OS's behavior, they require OS modification and therefore are not portable, or they incur high overheads thus perturbing the profiled OS.

We developed OSprof: a versatile, portable, and efficient OS profiling method based on latency distributions analysis. OSprof automatically selects important profiles for subsequent visual analysis. We have demonstrated that a suitable workload can be used to profile virtually any OS component. OSprof is portable because it can intercept operations and measure OS behavior from user-level or from inside the kernel without requiring source code. OSprof has typical CPU time overheads below 4%. In this paper we describe our techniques and demonstrate their usefulness through a series of profiles conducted on Linux, FreeBSD, and Windows, including client/server scenarios. We discovered and investigated a number of interesting interactions, including scheduler behavior, multi-modal I/O distributions, and a previously unknown lock contention, which we fixed.

1 Introduction

Profiling is a standard method to investigate and tune the operation of any complicated software component. Even the execution of one single-threaded user-level program is hardly predictable because of underlying hardware behavior. For example, branch prediction and cache behavior can easily change the program execution time by an order of magnitude. Moreover, there are a variety of possible external input patterns, and processes compete with each other for shared resources in multi-tasking environments. Therefore, only runtime profiling can clarify the actual system behavior even if the source code is available. At first glance, it seems that observing computer software and hardware behavior should not be difficult because it can be instrumented. However, profiling has several contradicting requirements: versatility, portability, and low overheads.

A versatile OS profile should contain information about the interactions between OS components and allow correlation of related information that was captured at different levels of abstraction. For example, a file system operates on files, whereas a hard disk driver operates on data blocks. However, the operation and performance of file systems and drivers depends on their complex interactions; contention on semaphores can change the disk's I/O patterns, while a file system's on-disk format dramatically changes its I/O behavior. This difficulty results in existing OS profiling tools that are hardly portable because they depend on a particular OS and hardware architecture. In addition, profilers for new kernels are often not available because existing profilers have to be ported to each new OS version. To minimize overheads, several hardware components provide profiling help. For example, modern CPUs maintain statistics about their operation [5]. However, only the OS can correlate this information with higher level information, such as the corresponding process. Therefore, some CPU time overheads are inevitable. To minimize these overheads, existing profilers provide limited information, can only profile specific types of activity, and rely on kernel instrumentation.

We developed a gray-box OS profiling method called OSprof. In an OS, requests arrive via system calls and network requests. The latency of these requests contains information about related CPU time, rescheduling, lock and semaphore contentions, and I/O delays. Capturing latency is fast and easy. However, the total latency includes a mix of many latencies contributed by different execution paths and is therefore difficult to analyze. Process preemption complicates this problem further. All existing projects that use latency as a performance metric use some simplistic assumptions applicable only in particular cases. Some authors assume that there is only one source of latency and characterize it using the average latency value [11, 12, 17, 30]. Others use prior knowledge of the latencies' sources to classify the latencies into several groups [3, 7, 27]. Other past attempts to analyze latencies more generally just look for distribution changes to detect anomalies [9]. Our profiling method allows latency investigation in the general case.

We compile the distributions of latencies for each OS operation, sort them into buckets at runtime, and later process the accumulated results. This allows us to ef-

ficiently capture small amounts of data that embody detailed information about many aspects of internal OS behavior. Different OS internal activities create different peaks on the collected distributions. The resulting information can be readily understood in a graphical form, aided by post-processing tools.

We created user-level profilers for POSIX-compliant OSs and kernel-level profilers for Linux and Windows, to profile OS activity for both local and remote computers. These tools have CPU time overheads below 4%. We used these profilers to investigate internal OS behavior under Linux, FreeBSD, and Windows. Under Linux, we discovered and characterized several semaphore and I/O contentions; source code availability allowed us to verify our conclusions and fix the problems. Under Windows, we observed internal lock contentions even without access to source code; we also discovered a number of harmful I/O patterns, including some for networked file systems.

The OSprof method is a general profiling and visualization technique that can be applied to a broad range of problems. Nevertheless, it requires skill to select proper workloads suitable for particular profiling goals. In this paper we describe several ways to select workloads and analyze corresponding profiles.

The rest of this paper is organized as follows. We describe prior work in Section 2. Section 3 describes our profiling method and provides some analysis of its applicability and limitations. Section 4 describes our implementation. We evaluate our system in Section 5. In Section 6 we present several usage scenarios and analyze profiles of several real-world file systems. We conclude in Section 7.

2 Background

The de facto standard of CPU-related code execution profiling is program counter sampling. Unix prof [4] instruments source code at function entry and exit points. An instrumented binary's program counter is sampled at fixed time intervals. The resulting samples are used to construct histograms with the number of individual functions invoked and their average execution times. Gprof [15] additionally records information about the callers of individual functions, which allows it to construct call graphs. Gprof was successfully used for kernel profiling in the 1980s [23]. However, the instrumented kernels had a 20% increase in code size and an execution time overhead of up to 25%. Kernprof [33] uses a combination of PC sampling and kernel hooks to build profiles and call graphs. Kernprof interfaces with the Linux scheduler to measure the amount of time that a kernel function spent sleeping in the profile (e.g., to perform I/O). Unfortunately, Kernprof requires a patch to both the kernel and the compiler, and overheads of 15% were reported. More detailed profiles with granularity as small as a single code line can be collected using *tcov* [34]. Most modern CPUs contain special hardware counters for use by profilers. The hardware counters allow correlation of profiled code execution, CPU cache states, branch prediction functionality, and ordinary CPU clock counts [2, 5]. Counter overflow events generate a non-maskable interrupt, allowing *Oprofile* [21] to sample events even inside device drivers (with overheads below 8%). Overall, such profilers are less versatile, capturing only CPU-related information.

There are a number of profilers for other aspects of OS behavior such as lock contention [6, 26]. They replace the standard lock-related kernel functions with instrumented ones. This instrumentation is costly: Lockmeter adds 20% system time overhead. Other specialized tools can profile memory usage, leaks, and caches [32].

Fewer and less developed tools are available to profile file system performance, which is highly dependent on the workload. Disk operations include mechanical latencies to position the head. The longest operation is seeking, or moving the head from one track to another. Therefore, file systems are designed to avoid seeks [24, 29]. Unfortunately, modern hard drives expose little information about the drive's internal data placement. The OS generally assumes that blocks with close logical block numbers are also physically close to each other on the disk. Only the disk drive itself can schedule the requests in an optimal way, and only the disk drive has detailed information about its internal operations. The Linux kernel optionally maintains statistics about the block-device I/O operations and makes those available through the /proc file system, yet little information is reported about timing.

Network packet sniffers capture traffic useful for system and protocol analysis [13]. Their problems are similar to those of hard disk profilers: both the client and server often perform additional processing that is not captured in the trace: searching caches, allocating objects, reordering requests, and more.

Latency contains important information and can be easily collected, but it cannot be easily analyzed because it likely contains a mix of latencies of different execution paths. Several profilers have used a simple assumption that there is one dominant latency contributor that can be characterized by the average latency [1,11,17]. This simple assumption allowed one to profile interrupts even on an idle system [14]. DeBox and LRP investigate average latency changes over time and their correlation with other system parameters [12, 30]. Chen and others moved one step further and observed changes in the distribution of latency over time and their correlation with software versions to detect possible problems in network services [9]. Prior knowledge of the under-

lying I/O characteristics and file system layouts allows categorization of runtime I/O requests based on their latency [3, 7, 27, 28].

There are several methods for integrating profiling into code. The most popular is direct source code modification because it imposes minimal overhead and is usually simple. For example, tracking lock contentions, page faults, or I/O activity usually requires just a few modifications to the kernel source code [6, 30]. If, however, every function requires profiling modifications, then a compiler-based approach may be more suitable (e.g., the gcc -p facility). More sophisticated approaches include runtime kernel code instrumentation and layered call interception. Solaris DTrace provides a comprehensive set of places in the kernel available for runtime instrumentation [8]. Dynamic code instrumentation is possible by inserting jump operations directly into the binary [16]. Similarly, debugging registers on modern CPUs can be used to instrument several code addresses at once [10]. Finally, stackable file systems may collect information about file system requests [37].

3 Profiler Design

OSs serve requests from applications whose workloads generate different request patterns. Latencies of OS requests consist of both CPU and wait times:

$$latency = t_{cpu} + t_{wait} \tag{1}$$

CPU time includes normal code execution time as well as the time spent waiting on spinlocks:

$$t_{cpu} = \sum t_{exec} + \sum t_{spinlock}$$

Wait time is the time in which a process was not running on the CPU. It includes synchronous I/O time, time spent waiting on semaphores, and time spent waiting for other processes or interrupts that preempted the profiled request:

$$t_{wait} = \sum t_{I/O} + \sum t_{sem} + \sum t_{int} + \sum t_{preempt}$$

 $t_{preempt}$ is the time in which the process was waiting because it ran out of its scheduling quantum and was preempted. We will consider preemption in Section 3.3. We begin by discussing the non-preemptive OS case.

Every pattern of requests corresponds to a set of possible execution paths S. For example, a system call that updates a semaphore-protected data structure can have two paths: (1) if the semaphore is available $(latency_1 = t_{cpu_1})$, or (2) if it has to wait on the semaphore $(latency_2 = t_{cpu_2} + t_{sem})$.

In turn, each t_j is a function with its own distribution. We can generalize that the $latency_s$ of paths $s \in S$ consists of the sum of latencies of their components:

$$latency_s = \sum_{j} t_{s,j} \tag{2}$$

where j is the component, such as I/O of a particular type, program execution-path time, or one of the spin-locks or semaphores.

To find all $t_j \in T$ it is necessary to solve the system of linear Equations 2, which is usually impossible because $\|T\| \geq \|S\|$ (there are usually fewer paths than time components). Non-linear logarithmic filtering is a common technique used in physics and economics to select only the major sum contributors [22]. We used latency filtering to select the most important latency contributors t_{max} and filter out the other latency components δ :

$$log(latency) = log(t_{max} + \delta) \approx log(t_{max})$$

For example, for log_2 , even if δ is equal to t_{max} , the result will only change by 1. Most non-trivial workloads can have multiple paths for the same operation (e.g., some requests may wait on a semaphore and some may not). To observe multiple paths concurrently we store logarithms of latencies into buckets. Thus, a bucket b contains the number of requests whose latency satisfies:

$$b = \lfloor \log_{\frac{1}{r}}(latency) \rfloor \approx \lfloor r \times \log(t_{max}) \rfloor$$

A profile's bucket density is proportional to the resolution r. For efficiency, we always used r=1. However, r=2, for example, would double the profile resolution (bucket density) with a negligible increase in CPU overheads and doubled (yet small overall) memory overheads.

Figure 1 shows an actual profile of the FreeBSD 6.0 clone operation called concurrently by four processes on a dual-CPU SMP system. We used CPU cycles as a time metric because it is the most precise and efficient metric available at run-time. For reference, the labels above the profile give the average buckets' latency in seconds. The y-axis shows the number of operations whose latency falls into a given bucket (note that both axes are logarithmic). In Figure 1, the two peaks correspond to two paths of the clone operation: (1) the left peak corresponds to a path without lock contention, and (2) the right peak corresponds to a path with a lock contention. Next, we will discuss how we collect and analyze the captured profiles.

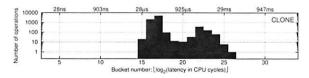


Figure 1: A profile of FreeBSD 6.0 clone operations concurrently issued by four user processes on a dual-CPU SMP system. The right peak corresponds to lock contention between the processes. Note: both axes are logarithmic (x-axis is base 2, y-axis is base 10).

3.1 Profile Collection and Analysis

Let us consider the profile shown in Figure 1 in more detail. We captured this profile entirely from the user level. In addition to the profile shown in Figure 1, we captured another profile (not shown here) with only a single process calling the same clone function; we observed that in that case there was only one (leftmost) peak. Therefore, we can conclude that there is some contention between processes inside of the clone function. In addition, we can derive information about (1) the CPU time necessary to complete a clone request with no contention (average latency in the leftmost peak) and (2) the portion of the clone code that is executed while a semaphore or a lock is acquired (average latency in the leftmost peak times the ratio of elements in the rightmost and leftmost buckets). In general, we use several methods to analyze profiles.

Profile preprocessing. A complete profile may consist of dozens of profiles of individual operations. For example, a user-mode program usually issues several system calls, so a complete profile consists of several profiles of individual system calls. If the goal of profiling is performance optimization, then we usually start our analysis by selecting a subset of profiles that contribute the most to the total latency. We designed automatic procedures to (1) select profiles with operations that contribute the most to the total latency; and (2) compare two profiles and evaluate their similarity. The latter technique has two applications. First, it can be used to compare all profiles in a complete set of profiles and select only these profiles that are correlated. Second, it is useful to compare two different complete sets of profiles and select only these pairs that differ substantially; this helps developers narrow down the set of OS operations where optimization efforts may be most beneficial. We have adopted several methods from the fields of statistics and visual analytics [31]. We further describe these methods in Section 3.2 and evaluate them in Section 5.3.

Prior knowledge-based analysis. Many OS operations have characteristic times. For example, we know that on our test machines, a context switch takes approximately $56\mu s$, a full stroke disk head seek takes approximately 8ms, a full disk rotation takes approximately 4ms, the network latency between our test machines is about $112\mu s$, and the scheduling quantum is about 58ms. Therefore, if some of the profiles have peaks close to these times, then we can hypothesize right away that they are related to the corresponding OS activity. For any test setup, these and many other characteristic times can be measured in advance by profiling simple workloads that are known to show peaks corresponding to these times. It is common that some peaks analyzed for one workload in one of the OS configurations can

be recognized later on new profiles captured under other circumstances.

Differential profile analysis. While analyzing profiles, one usually makes a hypothesis about a potential reason for a peak and tries to verify it by capturing a different profile under different conditions. For example, a lock contention should disappear if the workload is generated by a single process. The same technique of comparing profiles captured under modified conditions (including OS code or configuration changes) can be used if no hypothesis can be made. However, this usually requires exploring and comparing more sets of profiles. As we have already described in this section, we have designed procedures to compare two different sets of profiles automatically and select only those that differ substantially. Section 3.2 discusses these profiles, comparing procedures in more detail.

Layered profiling. It is usually possible to insert latency-profiling layers inside the OS. Most kernels provide extension mechanisms that allow for the interception and capture of information about internal requests. Figure 2 shows such an infrastructure. The inserted layers directly profile requests that are not coming from the user level (e.g., network requests). Comparison of the profiles captured at different levels can make the identification of peaks easier and the measurements more precise. For example, the comparison of user-level and filesystem-level profiles helps isolate VFS behavior from the behavior of lower file systems. Note that we do not have to instrument every OS component. For example, we will show later in this section that we can use file system instrumentation to profile scheduler or timer interrupt processing. Unlike specialized profilers, OSprof does not require instrumentation mechanisms to be provided by an OS, but can use them if they are available.

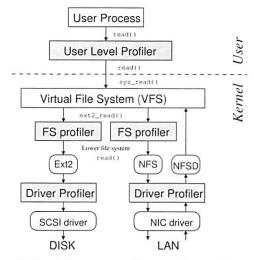


Figure 2: Our infrastructure allows profiling at the user, file system, and driver levels.

Layered profiling can be extended even to the granularity of a single function call. This way, one can capture profiles for many functions even if these functions call each other. To do so, one may instrument function entry and return points manually—for example, using the gcc -p facility. Similarly, many file system operations call each other. For example, the readdir operation of Linux 2.6 Ext2 calls the readpage operation if directory information is not found in the cache. Therefore, file-system-level profiling can also be considered to be a kind of layered profiling.

Profile sampling. OSprof is capable of taking successive snapshots by using new sets of buckets to capture latency at predefined time intervals. In this case we are also comparing one set of profiles against another, as they progress in time. This type of three-dimensional profiling is useful when observing periodic interactions or analyzing profiles generated by non-monotonic workload generators (e.g., a program compilation).

Direct profile and value correlation. If layered profiling is used, it is possible to correlate peaks on the profiles directly with the internal OS state. To do this, we first capture our standard latency profiles. Next, we sort OS requests based on the peak they belong to, according to their measured latency. We then store logarithmic profiles of internal OS parameters in separate profiles for separate peaks. In many cases, this allows us to correlate the values of internal OS variables directly with the different peaks, thus helping to explain them.

We will illustrate all of the above profile analysis methods in Section 6.

3.2 Automated Profile Analysis

We developed an automated profile analysis tool which (1) sorts individual profiles of a complete profile according to their total latencies; (2) compares two profiles and calculates their degree of similarity; and (3) performs these steps on two complete sets of profiles to automatically select a small set of "interesting" profiles for manual analysis.

The third step reflects our experience with using OS-prof. It operates in three phases. First, it ignores any profile pairs that have very similar total latencies, or where the total latency or number of operations is very small when compared to the rest of the profiles (the threshold is configurable). This step alone greatly reduces the number of profiles a person would need to analyze. In the second phase, our tool examines the changes between bins to identify individual peaks, and reports differences in the number of peaks and their locations. Third, we use one of several methods to rate the difference between the profiles.

Comparing two profiles. There are several methods for comparing histograms where only bins with the same index are matched. Some examples are the chisquared test, the Minkowski form distance [35], histogram intersection, and the Kullback-Leibler/Jeffrey divergence [20]. The drawback of these algorithms is that their results do not take factors such as distance into account because they report the differences between individual bins rather than looking at the overall picture.

The Earth Mover's Distance (EMD) is a cross-bin algorithm and is commonly used in data visualization as a goodness-of-fit test [31]. The idea is to view one histogram as a mass of earth, and the other as holes in the ground (the histograms are normalized so that we have exactly enough earth to fill the holes). The EMD value is the least amount of work needed to fill the holes with earth, where a unit of work is moving one unit by one bin. This algorithm does not suffer from the problems associated with the bin-by-bin and other cross-bin comparison methods, and is specifically designed for visualization. It indeed outperformed the other algorithms.

We also used two simple comparison methods: the normalized difference of total operations and of total latency. The algorithms are evaluated in Section 5.3.

3.3 Multi-Process Profiles

Capturing latency is simple and fast. However, early code-profiling tools rejected latency as a performance metric, because in multitasking OSs a program can be rescheduled at an arbitrary point in time, perturbing the results. We show here that rescheduling can reveal information about internal OS components such as the CPU scheduler, I/O scheduler, hardware interrupts, and periodic OS processes. Also, we will show conditions in which their influence on profiles can be ignored.

Forcible preemption effects. Execution in the kernel is different from execution in user space. Requests executed in the kernel usually perform limited amounts of computation. Some kernels (e.g., Linux 2.4 and Free-BSD 5.2) are non-preemptive and therefore a process cannot be rescheduled (though it can voluntarily yield the CPU, say, during an I/O operation or while waiting on a semaphore). Let us consider a fully preemptive kernel where a process can be rescheduled at any point in time. A process can be preempted during the profiled time interval only during its t_{cpu} component. Let Q be the quantum of time that a process is allowed to run by the scheduler before it is preempted. A process is never forcibly preempted if it explicitly yields the CPU before running for the duration of Q. This is the case in most of the practical scenarios that involve I/O or waiting on semaphores (i.e., yielding the CPU). Let Y be the probability that a process yields during a request. The probability that a process does not yield the CPU during Q

cycles is $(1-Y)^{\left(\frac{Q}{t_{period}}\right)}$, where t_{period} is the average sum of user and system CPU times between requests. If during Q cycles the process does not yield the CPU, then it will be preempted within the profiled time interval with probability $\frac{t_{cpu}}{t_{period}}$. Thus, the probability that a process is forcibly preempted while being profiled is:

$$Pr(fp) = \frac{t_{cpu}}{t_{period}} \times (1 - Y)^{\left(\frac{Q}{t_{period}}\right)}$$
 (3)

Differential analysis of Equation 3 shows that the function rapidly declines if $t_{period} \ll QY$. Plugging in our typical case numbers for times and 1% yield rate $(Y=0.01,t_{cpu}=\frac{t_{period}}{2}=2^{10},Q=2^{26})$ we get an extremely small forced preemption probability: 2.3×10^{-280} .

Figure 3 shows two profiles of read operation issued by two processes that were reading zero bytes of data from a file under Linux 2.6.11. One of the profiles was captured on a Linux kernel compiled with in-kernel preemption enabled (black bars) and the other one is captured with the kernel compiled with in-kernel preemption disabled (white bars). For easier comparison, both profiles are shown together. This workload has Y = 0and therefore can produce measurable preemption effects if we generate a large enough number of requests. We had to generate 2×10^8 requests to observe only 278 preempted requests in the 26^{th} bucket. This is consistent with our theory: the average latency of bucket b is equal to $t_{cpu} = \frac{3}{2}2^b$, and the expected number of preempted requests from bucket b is $n_b \frac{t_{epu}}{Q} = n_b \frac{\frac{3}{2}2^b}{Q}$, where n_b is the number of elements in bucket b. Summing up the expected number of preempted requests, we calculated that the expected number of elements in the 26th bucket is $388 \pm 33\%$ for Linux. We have also verified our theory for Windows XP and FreeBSD 6.0 on uniprocessor and SMP systems. We conclude that preemption effects can be ignored for all the profiles presented in this paper.

Profiles that contain a large number of requests also show information about low-frequency events (e.g., hardware interrupts or background OS threads) even if

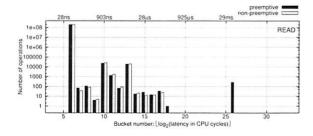


Figure 3: Profile of a read operation that reads zero bytes of data on a Linux 2.6.11 kernel compiled with in-kernel preemption enabled and the same kernel with preemption disabled.

these events perform a minimal amount of activity. Several small peaks in Figure 3 correspond to such activity. For example, on Linux the total duration of the profiling process divided by the number of elements in bucket 13 is equal to 4ms, which suggests that this peak corresponds to timer interrupt processing. Higher-resolution profiles may help analyze these peaks.

Wait times at high CPU loads. We normally assume that t_{wait} is defined by particular events such as I/O or a wait on a semaphore. However, if the CPU is still busy servicing another process after the t_{wait} time, then the request's latency will be longer than the original latency in Equation 1. Such a profile will still be correct because it will contain information about the affected t_{wait} . However, it will be harder to analyze as it will be shifted to the right; because the buckets are logarithmic, multiple peaks can become indistinguishable. Fortunately, this can happen only if the sum of the CPU times of all other processes is greater than t_{wait} .

3.4 Multi-CPU Profiles

There are two things one should keep in mind while profiling multi-CPU systems.

Clock Skew. CPU clock counters on different CPUs are usually not precisely synchronized. Therefore, the counters difference will be added to the measured latency if a process is preempted and rescheduled to a different CPU. Our logarithmic filtering produces profiles that are insensitive to counter differences that are less than the scheduling time. Fortunately, most systems have small counter differences after they are powered up (≈ 20 ns). Also, it is possible to synchronize the counters in software by writing to them concurrently. For example, Linux synchronizes CPU clock counters at boot time and achieves timing synchronization of ≈ 130 ns.

Profile Locking. Bucket increment operations are not atomic by default on most CPU architectures. This means that if two threads attempt to update the same bucket concurrently only one of them will succeed. A naïve solution would be to use atomic memory updates (the lock prefix on i386). Unfortunately, this can seriously affect profiler performance. Therefore, we adopted two alternative solutions based on the number of CPUs: (1) If the number of CPUs is small, the probability that two or more bucket writes happen at the same time is small. Therefore, the number of missed profile updates is small. For example, in the worst case scenario for a dual-CPU system, we observed that less than 1% of bucket updates were lost while two threads were concurrently measuring latency of an empty function and updating the same bucket. For real workloads this number is much smaller because the profiler updates different buckets and the update frequency is smaller. Therefore, we use *no* locking on systems with few CPUs. (2) The probability of concurrent updates grows rapidly as the number of CPUs increases. On systems with many CPUs we make each process or thread update its own profile in memory. This *prevents* lost updates on systems with any number of CPUs.

3.5 Method Summary

Our profiling method reveals useful information about many aspects of internal OS behavior. In general, profilers can be used to investigate known performance problems that are seen in benchmarks or during normal use, or to actively search for bottlenecks. We have used our profiler successfully in both ways.

When searching for potential performance issues, we found that a custom workload is useful to generate a profile that highlights an interesting behavior. In general, we start with simple workloads and devise more specific, focused workloads as the need arises. Workload selection is a repetitive-refinement visualization process, but we found that a small number of profiles tended to be enough to reveal highly useful information. We derived several formulas that allowed us to estimate the effects of preemption. We showed that for typical workloads (moderate CPU use and small number of system calls) preemption effects are negligible. Conversely, a different class of workloads (lots of CPU time and a large number of system calls) can expose preemption effects. This is useful when deriving the characteristics of internal OS components such as the CPU scheduler, I/O scheduler, and background interrupts and processes. Such information cannot be easily collected using other methods. While creating the workloads, one should keep in mind that workloads generated by many active processes can have high CPU loads and right-shift the latency peaks associated with I/O or semaphore activity. Fortunately, we found that just a few processes can already reveal most process-contention scenarios.

We do not require source code access, which enables us to perform gray-box profiling. The resulting profiles show which process or operation causes contention with another operation. For example, the profiles do not show which particular lock or semaphore is causing a slowdown, because that information is specific to a particular OS and therefore conflicts with our portability goal. Fortunately, the level of detail one can extract with the OSprof profiles is sufficient in most cases. For example, as we show in Section 6, the information we get is sufficient to find out which particular semaphore or lock is problematic if the source code is available. However, if the source code is not available one can still glean many details about a particular lock contention—enough to optimize an OS component or application that does not use the related lock directly.

When profiling from outside the kernel, OSprof does not add overheads to the kernel, and therefore has minimal impact on the OS's internal behavior. Moreover, OSprof traces requests at the interface level and adds small CPU-time overheads only on a per-request basis—without adding any overhead for each internal event being profiled (e.g., taking a semaphore).

4 Implementation

We designed a fast and portable aggregate_stats library that sorts and stores latency statistics in logarithmic buckets. Using that library, we created user-level, file-system—level, and driver-level profilers for Linux, FreeBSD, and Windows, as shown in Figure 2.

Instrumenting Linux 2.6.11 and FreeBSD 6.0 allowed us to verify some of the results by examining the source code. Instrumenting Windows XP allowed us to observe its internal behavior, which is not otherwise possible without access to the source code. We chose source code instrumentation techniques for Linux and FreeBSD for performance and portability reasons. We chose plugin or binary rewriting instrumentation for the Windows profilers because source code is not currently available.

The aggregate_stats library. This C library provides routines to allocate and free statistics buffers, store request start times in context variables, calculate request latencies, and store them in the appropriate bucket. We use the CPU cycle counter (TSC on x86) to measure time because it has a resolution of tens of nanoseconds, and querying it uses a single instruction. The TSC register is 64 bit wide and can count for a century without overflowing even for modern CPUs. To be consistent, we store all time values in terms of CPU cycles.

POSIX user-level profilers. We designed our user-level profilers with portability in mind. We directly instrumented the source code of several programs used to generate test workloads in such a way that system calls are replaced with macros that call our library functions to retrieve the value of the CPU timer, execute the system call, and then calculate the latency and store it in the appropriate bucket. This way, the same programs can be recompiled for other POSIX-compliant OSs and be used for profiling immediately. Upon exit, the program prints the collected profiles to the standard output.

Windows user-level profilers. To profile Windows under workloads generated by arbitrary non-open-sourced programs, we created a runtime system-call profiler. It is implemented as a DLL and uses the Detours library [16] to insert instrumentation functions for each system call of interest. Detours can insert new instrumentation into arbitrary Win32 functions even during program execution. It implements this by rewriting the target function images. To produce a workload, we ran a

Figure 4: Ext2 directory operations. The kernel exports the generic_read_dir function for use by many file systems.

program that executes the test application and injects the system call profiler DLL into the test program's address space. On initialization, the profiler inserts instrumentation functions for the appropriate Windows system calls.

Linux and FreeBSD file-system-level profilers. chose source code instrumentation to insert latency measurement code into existing Linux and FreeBSD file systems because it is simple and adds minimal profiling overheads. We implemented the profiling code as a FiST [37] file system extension. Our automatic code instrumentation system, FoSgen [18], parses this extension and applies it to file systems. It performs four steps: (1) Scan all source files for VFS operation vectors. (2) Scan all source files for operations found in the previous step and insert latency calculation macros in the function body; this also replaces calls to outside functions with wrappers. (3) Include a header file to declare the latency calculation macros in every C file that needs them. (4) Create aggregate_stats and /proc interface source files and add them to the Makefile.

FoSgen assumes that the file system's VFS operations are defined within fixed operation vectors. In particular, every VFS operation is a member of one of several data structures (e.g., struct inode_operations). These data structures contain a list of operations and hard-coded associated functions. For example, Figure 4 shows the definition of Ext2's file operations for directories. FoSgen discovers implementations of all file system operations and inserts FSPROF_PRE(op) and FSPROF_POST(op) macros at their entry and return points. For non-void functions of type f_type, FoSgen transforms statements of the form return foo(x) to:

```
f_type tmp_return_variable = foo(x);
FSPROF_POST(op);
return tmp_return_variable;
```

Often, file systems use generic functions exported by the kernel. For example, Ext2 uses the generic_read_dir kernel function for its read operation, as shown in Figure 4. FoSgen creates wrapper functions for such operations and instruments them to measure the latency of external functions.

FoSgen consists of 607 lines of perl code. Despite its simplicity, it successfully instrumented more than a dozen Linux 2.4.24, 2.6.11, and FreeBSD 6.0 file systems we tried it on. Also, FoSgen instrumented nullfs

and Wrapfs [37]—stackable file systems that can be mounted on top of other file systems to collect their latency profiles.

In addition to FoSgen, we have also created a simpler bash and sed script that can instrument Linux 2.4 and 2.6 file systems. The shell script contains 307 lines and 184 distinct sed expressions.

Windows file-system-level profilers. The Windows kernel-mode profiler is implemented as a file system filter driver [25] that stacks on top of local or remote file systems. In Windows, an OS component called the I/O Manager defines a standard framework for all drivers that handle I/O. The majority of I/O requests to file systems are represented by a structure called the I/O Request Packet (IRP) that is received via entry points provided by the file system. The type of an I/O request is identified by two IRP fields: MajorFunction and MinorFunction. In certain cases, such as when accessing cached data, the overhead associated with creating an IRP dominates the cost of the entire operation, so Windows supports an alternative mechanism called Fast I/O to bypass intermediate layers. Our file system profiler intercepts all IRPs and Fast I/O traffic that is destined to local or remote file systems.

Driver-level profilers. In Linux, file system writes and asynchronous I/O requests return immediately after scheduling the I/O request. Therefore, their latency contains no information about the associated I/O times. To detect this information, we instrumented a SCSI device driver; to do so we added four calls to the aggregate_stats library. Windows provides a way to create stackable device drivers, but we did not create one because the file system layer profiler already captures latencies of writes and asynchronous requests.

Representing results. We wrote several scripts to generate formatted text views and Gnuplot scripts to produce 2D and 3D plots. All the figures representing profiles in this paper were generated automatically. In addition, these scripts check the profiles for consistency, aggregate_stats maintains checksums of the number of time measurements. For every operation, results in all of the buckets are summed and then compared with the checksums. This verification catches potential code instrumentation errors.

Portability. Each of our instrumentation systems consists of three parts: (1) the aggregate statistics library, which is common to all instrumentation systems; (2) the instrumentation hooks; and (3) a reporting infrastructure to retrieve buckets and counts. Our aggregate statistics library is 141 lines of C code, and only the instruction to read the CPU cycle counter is architecture-specific. In the Linux kernel, we used the /proc interface for

reporting results, which consists of 163 lines. The instrumentation hooks for our Linux device driver used 10 lines. For user-space on Unix, our instrumentation and reporting interface used 68 lines.

Our Windows filter driver was based on FileMon [36] and totaled 5,262 lines, of which 273 were our own C code and 63 were string constants. We also wrote a 229-line user application to retrieve the profile from the kernel. We used the Detours library [16] for the Windows user-space profiling tool. We added 457 lines of C code to intercept 112 functions, of which 337 lines are repetitive pre-operation and post-operation hooks.

In sum, our profiling system is fairly portable, with less than 1,000 lines of code written for each of the three OSs. The aggregate statistics library runs without changes in four different environments: Unix applications, Windows applications, and the Unix and Windows kernels. We developed an automatic instrumentation tool for Linux and FreeBSD file systems that could be easily adapted for other Unix file systems.

5 Evaluation

We evaluated the overhead of our profiler using an instrumented Linux 2.6.11 Ext2 file system. We measured memory usage, CPU cache usage, the latency added to each profiled operation, and the overall execution time. We chose to instrument a file system, instead of a program, because a file system receives a larger number of requests (due to the VFS calling multiple operations for some system calls) and this demonstrates higher overheads. Moreover, user-level profilers primarily add overheads to user time. We conducted all our experiments on a 1.7GHz Pentium 4 machine with 256KB of CPU cache and 1GB of RAM. It uses an IDE system disk, but the benchmarks ran on a dedicated Maxtor Atlas 15,000 RPM 18.4GB Ultra320 SCSI disk with an Adaptec 29160 SCSI controller. We unmounted and remounted all tested file systems before each benchmark run. We also ran a program we wrote called chill that forces the OS to evict unused objects from its caches by allocating and dirtying as much memory as possible. We ran each test at least ten times and used the Student-t distribution to compute the 95% confidence intervals for the mean elapsed, system, user, and wait times. Wait time is the elapsed time less CPU time and consists mostly of I/O, but process scheduling can also affect it. In each case, the half-widths of the confidence intervals were less than 5% of the mean.

5.1 Memory Usage and Caches

We evaluated the memory and CPU cache overheads of the file system profiler. The memory overhead consists of three parts. First, there is some fixed overhead for the aggregation functions. The initialization functions are seldom used, so the only functions that affect caches are the instrumentation and sorting functions which use 231 bytes. This is below 1% of cache size for all modern CPUs. Second, each VFS operation has code added at its entry and exit points. For all of the file systems we tested, the code-size overhead was less than 9KB, which is much smaller than the average memory size of modern computers. The third memory overhead comes from storing profiling results in memory. A profile occupies a fixed memory area. Its size depends on the number of implemented file system operations and is usually less than 1KB.

5.2 CPU Time Overheads

To measure the CPU-time overheads, we ran Postmark v1.5 [19] on an unmodified and on an instrumented Ext2. Postmark simulates the operation of electronic mail servers. It performs a series of file system operations such as create, delete, append, and read. We configured Postmark to use the default parameters, but we increased the defaults to 20,000 files and 200,000 transactions so that the working set is larger then OS caches and so that I/O requests will reach the disk. This configuration runs long enough to reach a steady-state and it sufficiently stresses the system. Overall, the benchmarks showed that wait and user times are not affected by the added code. The unmodified Ext2 used 18.3 seconds of system time, or 16.8% of elapsed time. The instrumentation code increased system time by 0.73 seconds (4.0%). As seen in Figure 5, there are three additional components added: making function calls, reading the TSC register, and storing the results in the correct buckets. To understand the details of this per-operation overheads, we created two additional file systems. The first contains only empty profiling function bodies, to measure the overhead of calling the profiling functions. Here, the system time increase over Ext2 was 0.28 seconds (1.5%). The second file system read the TSC register, but did not include code to sort the information or store it into buckets. Here, the system time increased by 0.36 seconds over Ext2 (2.0%). Therefore, 1.5% of system

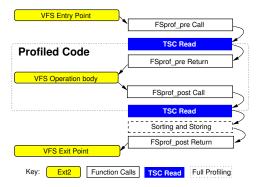


Figure 5: Profiled function components.

time overheads were due to calling profiling functions, 0.5% were due to reading the TSC, and 2.0% were due to sorting and storing profile information.

Not all of the overhead is included within the profile results. Only the portion between the TSC register reads is included in the profile, and therefore it defines the minimum value possible to record in the buckets. Assuming that an equal fraction of the TSC is read before and after the operation is counted, the delay between the two reads is approximately equal to half of the overhead imposed by the file system that only reads the TSC register. We computed the average overhead to be 40 cycles per operation. This result is confirmed by the fact that the smallest values we observed in any profile were always in the 5^{th} bucket. The 40-cycle overhead is well below most operation latencies, and can influence only the fastest of VFS operations that perform very little work. For example, sync_page is called to write a dirty page to disk, but it returns immediately if the page is not dirty. In the latter case its latency is at least 80 cycles long.

5.3 Automated Profile Analysis Accuracy

We conducted an informal study to measure the accuracy of our automated analysis tool. Three graduate students with twelve combined years of file system experience and six combined years of experience using OSprof examined over 250 profile pairs to determine which profiles contained important information (those which should be reported by an automated tool). We define a false positive as the tool reporting a profile that we did not consider to be important, and a false negative as the tool failing to report an important profile. We also sorted the same 250 profile pairs using our automated methods described in Section 3.2. The Chi-square method produced 5% of false positives and negatives; the total operation counts method produced 4%; the total latency method—3%; and the Earth Mover's Distance method had the smallest false classification rate of 2%.

6 Example File System Profiles

In this section we describe a few interesting examples that illustrate our method of analyzing OS behavior. We also illustrate our profile analysis methods described in Section 3.1. To conserve space, we concentrate on profiles of disk-based file systems and network file systems. Such profiles tend to contain a wide spectrum of events. We conducted all experiments on the same hardware setup as described in Section 5. Unless noted otherwise, we profiled a vanilla Linux 2.6.11 kernel and Windows XP SP2. All profiles presented in this section are from the file-system level except Figure 10.

We ran two workloads to capture the example profiles: a grep and a random-read on a number of file systems.

The grep workload was generated by the grep utility that was recursively reading through all of the files in the Linux 2.6.11 kernel source tree. The random-read workload was generated by two processes that were randomly reading the same file using direct I/O mode. In particular, these processes were changing the file pointer position to a random value and reading 512 bytes of data at that position. Of note is that we did not have to use many workloads to reveal a lot of new and useful information. After capturing just a few profiles, we were able to spot several problematic patterns.

6.1 Analyzing Disk Seeks

We ran the random-read workload generated by one and two processes. The automated analysis script alerted us to significant discrepancies between the profiles of the llseek operations. As shown in Figure 6, there are three interesting facts about the reported profiles that indicated process contention. First, the behavior did not exist when running with one process. Second, the 11seek operation was among the main latency contributors. This was noted as a strange behavior because the operation only updates the current file pointer position stored in the file data structure. Third, the right-most peak was strikingly similar with the read operation. Although these facts indicate that the llseek operation of one process competes with the read operation of the other process, llseek updates a per-process data structure, so we were unsure as to why that would happen.

Upon investigation of the source code, we verified that the delays were indeed caused by the <code>i_sem</code> semaphore in the Linux-provided method <code>generic_file_llseek</code>—a method which is used by most of the Linux file systems including Ext2 and Ext3. We observed that this contention happens 25% of the time, even with just two processes randomly reading the same file. We modified the kernel code to resolve this issue as follows. In particular, we observed that to be consistent with the semantics of other Linux VFS methods, we need only protect directory objects and not file objects. The <code>llseek</code> profile captured on the modified

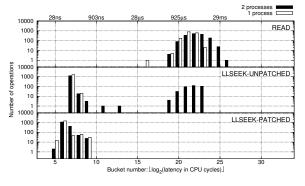


Figure 6: The 11seek operation under random reads.

kernel is shown at the bottom of Figure 6. As we can see, our fix reduced the average time of the llseek from 400 cycles to 120 cycles (a 70% reduction). The improvement is compounded by the fact that all semaphore and lock-related operations impose relatively high overheads even without contention, because the semaphore function is called twice and its size is comparable to llseek. Moreover, semaphore and lock accesses require either locking the whole memory bus or at least purging the same cache line from all other processors, which can hurt performance on SMP systems. (We submitted a small patch which fixes this problem and its description to the core Linux file system developers, who agreed with our reasoning and conclusions.)

We ran the same workload on a Windows NTFS file system and found no lock contention. This is because keeping the current file position consistent is left to user-level applications on Windows.

6.2 Analyzing File System Read Patterns

We now show how we analyzed various file system I/O patterns under the grep workload. In this workload, we use the grep utility to search recursively through the Linux kernel sources for a nonexistent string. Most of the peaks shown in the top profile of Figure 7 are common for all file system operations that require hard disk accesses. Here we refer to the readdir operation peaks by their order from left to right: first (buckets 6 and 7), second (9–14), third (16 and 17), and fourth (18–23).

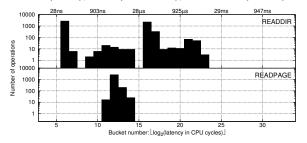


Figure 7: Profiles of Linux 2.6.11 Ext2 readdir (top) and readpage (bottom) operations captured for a single run of grep -ron a Linux source tree.

First peak (buckets 6–7). From the profile of Figure 3 we already know that on Linux the peak in the 6^{th} bucket corresponds to a read of zero bytes of data or any other similar operation that returns right away. The readdir function returns directory entries in a buffer beginning from the current position in the directory. This position is automatically updated when reading and modifying the directory and can also be set manually. If the current position is past the end of the directory, readdir returns immediately (this happens when a program repeatedly calls readdir until no more entries are returned). Therefore, it seems likely that the first peak corresponds

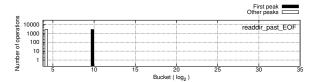


Figure 8: Correlation of the readdir_past_EOF $\times 1,024$ and the requests' peaks in Figure 7.

to the reads past the end of directory. One way to verify our hypothesis would be to profile a workload that issues readdir calls only after there are no more directory entries to read and then compares the resulting profiles (differential analysis). However, we can demonstrate our other method of profile analysis by directly correlating peaks and variables.

To do so, we slightly modified our profiling macros: instead of storing the latency in the buckets we (1) calculated a readdir_past_EOF value for every readdir call (readdir_past_EOF = 1 if the file pointer position is greater or equal to the directory buffer size and is 0 otherwise); (2) if the latency of the current function execution fell within the range of the first peak, a value of the bucket corresponding to readdir_past_EOF $\times 1,024$ was incremented in one profile and in another profile otherwise. The resulting profiles are shown in Figure 8 and prove our hypothesis.

Second peak (buckets 9–14). The readdir operation calls the readpage operation for pages not found in the cache. The readpage profile is a part of the complete grep workload profile and is shown on the bottom in Figure 7. During the complete profile preprocessing phase our automatic profiles analysis tool discovered that the number of elements in the third and fourth peaks is exactly equal to the number of elements in the readpage profile. This immediately suggests that the second peak corresponds to readdir requests that were satisfied from the cache. Note that the latency of readpage requests is small compared to related readdir requests. That is because readpage just initiates the I/O and does not wait for its completion.

Third peak (buckets 16–17). The third and the fourth peaks of the readdir operation correspond to disk I/O requests. The third peak corresponds to the fastest I/O requests possible. It does not correspond to I/O requests that are still being read from the disk and thus may require disk rotations or even seeks. This is because the shape of the third peak is sharp (recall that the Y scale is logarithmic). Partially read data requests would have to wait for partial disk rotations and thus would spread to gradually merge with the fourth peak. Therefore, the third peak corresponds to I/O requests satisfied from the disk cache due to internal disk readahead.

Fourth peak (buckets 18–23). We know that the fourth peak corresponds to requests that may require seeking with a disk head (track-to-track seek time for our hard drive is 0.3ms; full stroke seek time is 8ms) and waiting for the disk platter to rotate (full disk rotation time is 4ms).

6.3 Reiserfs and Timeline Profiles

On Linux, atime updates are handled by the Linux buffer flushing daemon, bdflush. This daemon writes data out to disk only after a certain amount of time has passed since the buffer was released; the default is thirty seconds for data and five seconds for metadata. This means that every five and thirty seconds, file system behavior may change due to the influence of bdflush. We use our profile sampling method to analyze the behavior of such periodic behavior. A sampled profile is similar to our standard profile, but instead of adding up all of the operations for a given workload, we divide the profile into fixed-time segments and save each segment separately. This mode of operation is possible thanks to the small size of the OSprof profile data. In Figure 9, we show such a 3-dimensional profile of a known lock contention between write_super and read operations on Reiserfs 3.6 on Linux 2.4.24. The x-axis shows the bucket number and the y-axis shows the elapsed time in CPU cycles. The three vertical black stripes on the read profiles correspond to those peaks already shown in Figure 7: cached reads, disk buffer reads, and reads with a disk access.

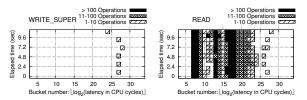


Figure 9: Linux 2.4.24 Reiserfs 3.6 file-system profiles sampled at 2.5 second intervals.

6.4 Analyzing Network File Systems

We connected two identical machines (described in Section 5) with a 100Mbps Ethernet link and ran several workloads. The purpose was to compare clients using the Linux implementation of the SMB protocol with those using the Windows implementation of the CIFS protocol (a modified version of SMB). The server ran Windows with an NTFS drive shared over CIFS. Our automated analysis script selected just six out of 51 profiled operations based on their total latency when running our grep workload. Among them, the FindFirst and FindNext operations on the Windows client had peaks that were farther to the right than any other operation (buckets 26–30 in the top two graphs of Fig-

ure 10). Layered profile analysis showed that these two peaks were not present in the profiles of the Linux client and alone account for 12% of the elapsed time, which was 170 seconds in total. FindFirst searches for file names with a given pattern and returns all matching file names along with their associated metadata information. It also returns a cookie, which allows the caller to continue receiving matches by passing it to FindNext.

By examining the peaks in other operations on the client (e.g., read shown in Figure 10) and the corresponding requests on the server, we found that instances of an operation which fall into bucket 18 and higher (greater than $168\mu s$) involve interaction with the server, whereas buckets to the left of it were local to the client. All of the FindFirst operations here go through the server, as do the rightmost two peaks of the FindNext operation. We ran a packet sniffer on the network to investigate this further.

A timeline for a typical FindFirst transaction between a Windows client and a Windows server explains the source of the problems and is shown on the lefthand side of Figure 11. The client begins by sending a FindFirst request containing the desired pattern to search for (e.g., $C: \lim_{\to \infty} 0.11 \times$). The server replies with file names that match this pattern and their associated metadata. Since the reply is too large for one TCP packet, it is split into three packets ("FIND_FIRST reply," "reply continuation 1," and "reply continuation 2"). The acknowledgment (ACK) for "reply continuation 1" is sent immediately, but the ACK for "reply continuation 2" is sent only after approximately 200ms. This behavior is a delayed ACK: because TCP can send an ACK in the same packet as other data, it delays the ACK in the hope that it will soon need to send another packet to the same destination. Most implementations wait 200ms for other data to be sent before sending an ACK on its own. Delaying an ACK is considered to be good behavior, but the Windows server does not continue to send data until it has received an ACK for everything until that point. This unnecessary synchronous behavior is what causes poor performance

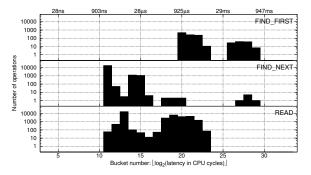


Figure 10: FindFirst, FindNext, and read operations on the Windows client over CIFS.

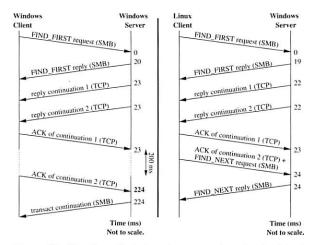


Figure 11: Timelines depicting the transactions involved in the handling of a FindFirst request between a Windows client and server over CIFS (left) and between a Linux client and a Windows server over SMB (right) as recorded on the server. Times are in milliseconds (not drawn to scale). Protocols are shown in parentheses.

for the FindFirst and FindNext operations. After the server receives this ACK, it sends the client a "transact continuation" SMB packet, indicating that more data is arriving. This is followed by more pairs of TCP replies and ACKs, with similar delays.

The right-hand side of Figure 11 shows a similar timeline for a Linux client interacting with a Windows server over SMB. The behavior is identical, except that instead of sending a delayed ACK for "reply continuation 2," Linux sends a FindNext request immediately that also contains an ACK. This causes the server to return more entries immediately. We modified a Windows registry key to turn off delayed ACKs, and found that it improved elapsed time by 20%. This is not a solution to the problem, but a way to approximate potential performance benefits without waiting on ACKs.

7 Conclusions

We designed a new OS profiling method that is versatile, portable, and efficient. The method allows the person profiling to consider and analyze the OS and the events being profiled at a high level of abstraction. In particular, the events can be anything that contribute to the OS execution latencies. Our method allows profiling various characteristics and behavior of the whole OS, including the I/O subsystem. The resulting profiles indicate pathologies and their dependencies. Access to the source code allows us to investigate these abstract characteristics such as lock or semaphore contentions. However, even without the source code, most of the problems can be described and studied in detail. While versatile, our method allows profiling with very high precision of about 40 CPU cycles and negligible overheads of only about 200 CPU cycles per profiled OS entry point.

When run with an I/O-intensive workload, we measured elapsed time overhead of less than 1%.

We used our method to collect and analyze profiles for task schedulers, CPU-bound processes, and several popular Linux, FreeBSD, and Windows file systems (Ext2, Ext3, Reiserfs, NTFS, and CIFS). To aid this analysis, we developed automatic processing and visualization scripts to present the results clearly and concisely. We discovered, investigated, and explained multi-modal latency distributions. We also identified pathological performance problems related to lock contention, network protocol inconsistency, and I/O interference.

Future work. We plan to apply our methods to more OSs and use higher resolution profiles to explain more complex internal OS behavior. Because of the compactness of our profiles, we believe that OSprof is suitable for clusters and distributed systems. We plan to expand OSprof for use on such large systems, explore scalability issues, and visual analytics driven profile automation.

Acknowledgments. We thank the anonymous program committee members and our shepherd Derek McAuley for their valuable comments. Also we thank Steve Skiena for recommending the Earth Mover's Distance algorithm. This work was partially made possible by NSF CAREER EIA-0133589, CCR-0310493, and HECURA CCF-0621463 awards and HP/Intel gifts numbers 87128 and 88415.1.

For more information and software, see

www.fsl.cs.sunysb.edu/project-osprof.html.

References

- M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. of the 19th* ACM Symposium on Operating Systems Principles, pp. 74–89, Bolton Landing, NY, October 2003.
- [2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proc.* of the 16th Symposium on Operating Systems Principles, pp. 1–14, Saint Malo, France, October 1997
- [3] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, pp. 43–56, Banff, Canada, October 2001
- [4] Bell Laboratories. prof, January 1979. Unix Programmer's Manual, Section 1.
- [5] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proc. of the 2000 ACM/IEEE conference on Super*computing, pp. 42–54, 2000.
- [6] R. Bryant and J. Hawkes. Lockmeter: Highlyinformative instrumentation for spin locks in the Linux

- kernel. In *Proc. of the 4th Annual Linux Showcase and Conf.*, pp. 271–282, Atlanta, GA, October 2000.
- [7] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proc. of the Annual USENIX Technical Conf.*, pp. 29–44, Monterey, CA, June 2002.
- [8] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proc.* of the Annual USENIX Technical Conf., pp. 15–28, 2004.
- [9] M. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *1st USENIX/ACM Symposium on Net*worked Systems Design and Implementation, pp. 309– 322, San Francisco, CA, March 2004.
- [10] W. Cohen. Gaining insight into the Linux kernel with Kprobes. *RedHat Magazine*, March 2005.
- [11] E. Cota-Robles and J. Held. A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98. In *Proc. of the Third Symposium on Oper*ating Systems Design and Implementation, pp. 159–172, New Orleans, LA, February 1999.
- [12] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In Proc. of the Second Symposium on Operating Systems Design and Implementation, pp. 261–275, Seattle, WA, October 1996.
- [13] D. Ellard and M. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proc. of the Annual USENIX Conf. on Large Installation Systems Administration*, San Diego, CA, October 2003.
- [14] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer. Using Latency to Evaluate Interactive System Performance. In Proc. of the Second Symposium on Operating Systems Design and Implementation, pp. 185–199, Seattle, WA, October 1996.
- [15] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *Proc. of the* 1982 SIGPLAN symposium on Compiler construction, pp. 120–126, June 1982.
- [16] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In Proc. of the 3rd USENIX Windows NT Symposium, July 1999.
- [17] M. Jones and J. Regehr. The Problems You're Having May not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT. In *Proc. of* the 1999 Workshop on Hot Topics in Operating Systems (HotOS VII), pp. 96–102, Rio Rico, AZ, March 1999.
- [18] N. Joukov and E. Zadok. Adding Secure Deletion to Your Favorite File System. In Proc. of the third international IEEE Security In Storage Workshop, San Fransisco, CA, December 2005.
- [19] J. Katcher. PostMark: A New Filesystem Benchmark. Tech. Rep. TR3022, Network Appliance, 1997. www.netapp.com/tech_library/3022.html.
- [20] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, March 1951.

- [21] J. Levon and P. Elie. Oprofile: A system profiler for linux. http://oprofile.sf.net, September 2004.
- [22] R. N. Mantegna and H. E. Stanley. An Introduction to Econophysics: Correlations and Complexity in Finance. Cambridge University Press, 2000.
- [23] M. K. McKusick. Using gprof to tune the 4.2BSD kernel. http://docs.freebsd.org/44doc/ papers/kerntune.html, May 1984.
- [24] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. ACM Transactions on Computer Systems, 2(3):181–197, August 1984.
- [25] Microsoft Corporation. File System Filter Manager: Filter Driver Development Guide. www.microsoft. com/whdc/driver/filterdrv/default.mspx, September 2004.
- [26] A. Morton. sleepometer. www.kernel.org/pub/ linux/kernel/people/akpm/patches/2.5/2.5.74/2.5. 74-mml/broken-out/sleepometer.patch, July 2003.
- [27] J. Nugent, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Controlling Your PLACE in the File System with Gray-box Techniques. In *Proc. of the Annual USENIX Technical Conf.*, pp. 311–323, San Antonio, TX, June 2003.
- [28] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea. Analysis and Evolution of Journaling File Systems. In *Proc. of the Annual USENIX Technical Conf.*, Anaheim, CA, May 2005.
- [29] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proc.* of 13th ACM Symposium on Operating Systems Principles, pp. 1–15, Asilomar Conf. Center, Pacific Grove, CA, October 1991.
- [30] Y. Ruan and V. Pai. Making the "Box" Transparent: System Call Performance as a First-class Result. In *Proc. of the Annual USENIX Technical Conf.*, pp. 1–14, Boston, MA, June 2004.
- [31] Y. Rubner, C. Tomasi, and L. J. Guibas. A Metric for Distributions with Applications to Image Databases. In Proc. of the Sixth International Conf. on Computer Vision, pp. 59–66, Bombay, India, January 1998.
- [32] J. Seward, N. Nethercote, and J. Fitzhardinge. Valgrind. http://valgrind.kde.org, August 2004.
- [33] Silicon Graphics, Inc. Kernprof (Kernel Profiling). http://oss.sgi.com/projects/kernprof, 2003.
- [34] Sun Microsystems. Analyzing Program Performance With Sun Workshop, February 1999. http://docs.sun.com/db/doc/805-4947.
- [35] M. J. Swain and D. H. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.
- [36] Sysinternals.com. Filemon. www.sysinternals.com/ ntw2k/source/filemon.shtml, 2004.
- [37] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the Annual USENIX Technical Conf.*, pp. 55–70, San Diego, CA, June 2000.

CRAMM: Virtual Memory Support for Garbage-Collected Applications

Ting Yang

Emery D. Berger

Scott F. Kaplan[†]

J. Eliot B. Moss

tingy@cs.umass.edu

emery@cs.umass.edu

sfkaplan@cs.amherst.edu

moss@cs.umass.edu

Dept. of Computer Science University of Massachusetts Amherst Amherst, MA 01003-9264 †Dept. of Mathematics and Computer Science Amherst College Amherst, MA 01002-5000

Abstract

Existing virtual memory systems usually work well with applications written in C and C++, but they do not provide adequate support for garbage-collected applications. The performance of garbage-collected applications is sensitive to heap size. Larger heaps reduce the frequency of garbage collections, making them run several times faster. However, if the heap is too large to fit in the available RAM, garbage collection can trigger thrashing. Existing Java virtual machines attempt to adapt their application heap sizes to fit in RAM, but suffer performance degradations of up to 94% when subjected to bursts of memory pressure.

We present CRAMM (Cooperative Robust Automatic Memory Management), a system that solves these problems. CRAMM consists of two parts: (1) a new virtual memory system that collects detailed reference information for (2) an analytical model tailored to the underlying garbage collection algorithm. The CRAMM virtual memory system tracks recent reference behavior with low overhead. The CRAMM heap sizing model uses this information to compute a heap size that maximizes throughput while minimizing paging. We present extensive empirical results demonstrating CRAMM's ability to maintain high performance in the face of changing application and system load.

1 Introduction

The virtual memory (VM¹) systems in today's operating systems provide relatively good support for applications written in the widely-used programming languages of the 80's and 90's, such as C and C++. To avoid the high overhead of heavy page swapping, it is sufficient for these applications to fit their working sets in physical memory [16]. VM systems typically manage physical memory memory with an approximation of a global LRU policy [12, 13, 15, 16, 22], which works reasonably well for legacy applications.

However, garbage-collected languages are now increasingly prevalent, ranging from general-purpose languages

like Java and C# to scripting languages like Python and Ruby. Garbage collection's popularity derives from its many software engineering advantages over manual memory management, including the elimination of dangling pointer errors and a drastic reduction of memory leaks.

The performance of garbage-collected applications is highly sensitive to heap size. A smaller heap reduces the amount of memory referenced, but requires frequent garbage collections that hurt performance. A larger heap reduces the frequency of collections, thus improving performance by up to 10x. However, if the heap cannot fit in available RAM, performance drops off suddenly and sharply. This is because garbage collection has a large working set (it touches the entire heap) and thus can trigger catastrophic page swapping that degrades performance and increases collection pauses by orders of magnitude [18]. Hence, heap size and main memory allocation need to be coordinated to achieve good performance. Unfortunately, current VM systems do not provide sufficient support for this coordination, and thus do not support garbagecollected applications well.

Choosing the appropriate heap size for a garbagecollected application—one that is large enough to maximize throughput but small enough to avoid paging-is a key performance challenge. The ideal heap size is one that makes the working set of garbage collection just fit within the process's main memory allocation. However, an a priori best choice is impossible in multiprogrammed environments, since the amount of main memory allocated to each process constantly changes. Existing garbagecollected languages either ignore this problem, allowing only static heap sizes, or adapt the heap size dynamically using mechanisms that are only moderately effective. For example, Figure 1 shows the effect of dynamic memory pressure on an industrial-strength Java virtual machine, BEA's JRockit [7], running a variant of the SPECjbb2000 benchmark. The solid line depicts program execution when given a fixed amount of RAM, while the dashed line shows execution under extra periodic bursts of memory pressure.

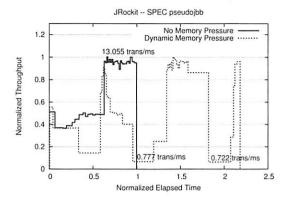


Figure 1: Impact of bursts of memory pressure on the performance on the JRockit Java virtual machine, which degrades throughput by as much as 94%.

This memory pressure dilates overall execution time by a factor of 220%, and degrades performance by up to 94%.

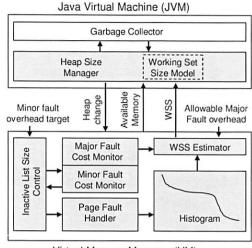
The problem with these adaptive approaches is not that their adaptivity mechanism is broken, but rather that they are *reactive*. The only way these systems can detect whether the heap size is too large is to grow the heap until paging occurs, which leads to unacceptable performance degradation.

Contributions: This paper makes the following contributions. It presents CRAMM (Cooperative Robust Automatic Memory Management), a system that enables garbage-collected applications to *predict* an appropriate heap size, allowing the system to maintain high performance while adjusting dynamically to changing memory pressure.

CRAMM consists of two parts; Figure 2 presents an overview. The first part is the CRAMM VM system that dynamically gathers the *working set size (WSS)* of each process, where we define the WSS as *the main memory allocation that yields a trivial amount of page swapping*. To accomplish this, CRAMM VM maintains separate page lists for each process and computes an *LRU reference histogram* [25, 27] that captures detailed reference information while incurring little overhead (around 1%).

The second part of CRAMM is its heap sizing model, which controls application heap size and is independent of any particular garbage collection algorithm. The CRAMM model correlates the WSS measured by the CRAMM VM to the current heap size. It then uses this correlation to select a new heap size that is as large as possible (thus maximizing throughput) while yielding little or no page faulting behavior. We apply the CRAMM model to five different garbage collection algorithms, demonstrating its generality.

We have implemented the CRAMM VM system in the Linux kernel and the CRAMM heap sizing model in



Virtual Memory Manager (VM)

Figure 2: The CRAMM system. The CRAMM VM system efficiently gathers detailed *per-process* reference information, allowing the CRAMM heap size model to choose an optimal heap size dynamically.

the Jikes RVM research Java virtual machine [3]. We present the results of an extensive empirical evaluation of CRAMM, including experimental measurements across 20 benchmarks and 5 garbage collectors, as well as comparison with two industrial Java implementations. These results demonstrate CRAMM's effectiveness in maintaining high performance in the face of changes in application behavior and system load.

This work builds on our previous study that introduced an early version of the CRAMM heap sizing model [28]. That study presented a model that was evaluated only in the context of trace-driven simulations. This paper builds on the previous study significantly. It refines the heap sizing model to take into account copying and non-copying regions (required to handle generational collectors), provides more accurate startup adjustment, and more effectively adapts to dynamic memory allocation by polling the underlying VM between collections. Furthermore, it is implemented in a fully functional kernel and JVM, introduces implementation strategies that make its overhead practical, has more efficient overhead control mechanisms, and presents extensive empirical results.

In addition to serving the needs of garbage-collected applications, the CRAMM VM system is the first system to our knowledge to provide per-process and per-file page management while efficiently gathering detailed reference histograms. This information can be used to implement a wide range of recently proposed memory management systems including compressed caching [27], adaptive LRU policies like EELRU [25], and informed prefetchers [20, 24].

The remainder of this paper is organized as follows. Section 2 presents an overview of garbage collection algorithms and terminology used in this paper. Section 3 derives the CRAMM heap sizing model, which relates application working set size to heap size. Section 4 describes the CRAMM VM system, which gathers detailed statistics allowing it to compute the precise current process working set size. Section 5 presents empirical results, comparing static and previous adaptive approaches to CRAMM. Section 6 presents work most closely related to ours, and Section 7 concludes.

2 GC Behavior and Terminology

A garbage collector (GC) periodically and automatically finds and reclaims heap-allocated objects that a program can no longer possibly use. We now sketch how, and when, a GC may do this work, and along the way introduce GC terminology and concepts critical to understanding CRAMM.

Garbage collectors operate on the principle that if an object is *unreachable* via any chain of pointers starting from *roots*—pointers found in global/static variables and on thread stacks—then the program cannot possibly use the object in the future, and the collector can reclaim and reuse the object's space. Through a slight abuse of terminology, reachable objects are often called *live* and unreachable ones *dead*. Reference counting collectors determine (conservatively) that an object is unreachable when there are no longer any pointers to it. Here, we focus primarily on *tracing collectors*, which actually trace through pointer chains from roots, visiting reachable objects.

The frequency of collection is indirectly determined by the *heap size*: the maximum virtual memory space that may be consumed by heap-allocated objects. When allocations have consumed more than some portion of the heap size (determined by the collection algorithm), collection is invoked. Thus, the smaller the heap size, the more frequently GC occurs, and the more CPU time is spent on collection.

GC algorithms divide the heap into one or more regions. A non-generational GC collects all regions during every collection, triggering collection when some percentage of the entire heap space is filled with allocated objects. A nongenerational GC may have only one region. In contrast, generational GCs partition the regions into groups, where each group of regions, called a generation, contains objects of a similar age. Most commonly, each group consists of a single region. When some percentage of the space set aside for a generation has been filled, that generation, and all younger ones, are collected. Additionally, live objects that survive the collection are generally promoted to the next older generation. New objects are typically allocated into a nursery region. This region is usually small, and thus is collected frequently, but quickly (because it is small). The generational configurations that we consider here have two generations, a nursery and a *mature space*. Because nursery collection generally filters out a large volume of objects that die young, mature space grows more slowly—but when it fills, that triggers a *full heap* collection.

Orthogonal to whether a collector is generational is how it reclaims space. *Mark-sweep (MS)* collection marks the reachable objects, and then sweeps across the allocation region to reclaim the unmarked ones. MS collection is *non-copying* in that it does not move allocated objects. In contrast, *copying* collectors proceed by copying reachable objects to an empty copy space, updating pointers to refer to the new copies. When done, it reclaims the previous copy space. We do not consider here collectors that compact in place rather than copying to a new region, but our techniques would work just as well for them. Notice that collectors that have a number of regions may handle each region differently. For example, a given GC may collect one region by copying, another by MS, and others it may never collect (so-called *immortal spaces*).

Finally, allocation and collection are intertwined. When allocating into an MS-managed region, the allocator may use free lists to find available chunks of space. When allocating into a copying region, it typically increments a free space pointer through the initially empty space. For generational collection, the nursery is usually a copy-collected space, thus allowing fast allocation. The mature space, however, may be a copying- or a non-copying-collected region, depending on the particular collector.

3 CRAMM Heap Sizing Model

The goal of the CRAMM heap sizing model is to relate heap size and working set size, so that, given a current real memory allocation, we can determine a heap size whose working set size just fits in the allocation. The working set size (WSS) for a GCed application is determined almost entirely by what happens during full collections, because full collections touch every reachable heap object. Since live and dead objects are generally mixed together, the working set includes all heap pages used for allocated objects. It also includes the space needed for copied survivors of copying regions. Thus, each non-copying region contributes its size to the working set, while each copying region adds its size plus the volume of copied survivors, which can be as much as the size of the copying region in the worst case.

Several properties of GCed applications are important here. First, given adequate real memory, performance varies with heap size. For example, Figure 3 depicts the effect of different amounts of memory (the size of the garbage-collected heap) on performance. This graph is for a particular benchmark and garbage collector (the SPECjvm98 benchmark javac with a mark-sweep garbage collector), but it is typical. On the left-hand side, where the heap is barely large enough to fit the applica-

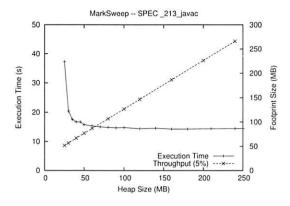


Figure 3: The effect of heap size on performance and working set size (the number of pages needed to run with at most a 5% slowdown from paging).

tion, execution time is high. As the heap size increases, execution time sharply drops, finally running almost 250% faster. This speedup occurs because a larger heap reduces the number of collections, thus reducing GC overhead. The execution time graph has a 1/x shape, with vertical and horizontal asymptotes.

However, the *working set size*—here given as the amount of memory required to run with at most 5% elapsed time added for paging—has a linear shape. The heap size determines the working set size, as previously described. Our earlier work explores this in more detail [28]. The key observation is that working set size is very nearly linear in terms of heap size.

3.1 GC Working Set Size and Heap Sizing Model

We define heap size, H, as the maximum amount of space allowed to contain heap objects (and allocation structures such as free lists) at one time. If non-copy-collected regions use N pages and copy-collected regions allocate objects into C pages, then $H = N + 2 \times C$. We must reserve up to C pages into which to copy survivors from the original C space, and the collector needs both copies until it is done. The total WSS for the heap during full collection is determined by the pages used for copied survivors, CS: WSS = N + C + CS. Thus, heap WSS varies from N + C to $N + 2 \times C$.

As a program runs, its usage of non-copying and copying space may vary, but it is reasonable to assume that the balance usually does not change rapidly from one full collection to the next. We call the ratio of allocable space (N+C) to heap size $(N+2\times C)$ the heap utilization, u. It varies from 50% for N=0 to 100% for C=0. Given an estimate of u, we can determine N+C from H, but to determine WSS, we also need to estimate CS. Fortunately, CS is a property of the application (volume of live objects in copy-collected regions), not of the heap size. As with u, we can reasonably assume that CS does not change too

rapidly from one full collection to the next.

When adjusting the heap size, we use this equation as our model: $\Delta H = (\Delta WSS - \Delta CS)/u$. Notice that ΔWSS is just our target WSS (i.e., the real memory allocation the OS is willing to provide) minus our current WSS. The CRAMM VM provides both of these values to the heap size manager.

Starting out: Once the JVM reaches the point where it needs to calculate an initial heap size, it has touched an initial working set of code and data. Thus, the space available for the heap is exactly the volume of free pages the VM system is willing to grant us (call that *Free*). We wish to set our heap size so that our worst case heap WSS during the first full collection will not exceed *Free*. But the worst heap WSS is exactly the heap size, so we set *H* to the minimum of *Free* and the user-requested initial heap size.

Tracking the parameters: To determine the heap utilization u, we simply calculate it at the end of each collection, and assume that the near future will be similar. Estimating ΔCS is more involved. We track the maximum value for CS that we have seen so far, maxCS, and we also track the maximum increment we have seen to CS, maxC-SInc. If, after a full collection, CS exceeds maxCS, we assume CS is increasing and estimate $\Delta CS = maxCSInc/2$, i.e., that it will grow by 1/2 of the largest increment. Otherwise we estimate ΔCS as maxCS - CS, i.e., that CS for the next full collection will equal maxCS. After calculating ΔCS , we decay maxCS, multiplying it by 0.98 (a conservative policy to maintain stable maxCS), and maxCSInc, multiplying it by 0.5 (a more rapidly adjusting policy so that maxCSInc decays away quickly once maxCS reaches its stable state).

Handling nursery collections: Because nursery collections do not process the whole heap, their CS value underestimates survival from future full collections. So, if the nursery size is less than 50% of allocable space, we do not update H. For larger nurseries, we estimate ΔCS by multiplying the size of uncollected copying space times $1 + \sigma$, where σ is the *survival rate* of the nursery collection, i.e., CS/v, where v is the size of the nursery.

This model is a straightforward generalization of our previous one [28], taking into account copying and non-copying regions and modeling startup effects, and eliminates the overhead (8% - 23%) caused by inaccurate startup adjustment in our previous model. Tracking of *maxCS* and *maxCSInc* also helps avoid paging. We periodically request the current *Free* value on allocation slow path, when the allocator tries to request a new chunk of memory from the VM (128KB for MS and 1MB for others). Once *Free* is less than *maxCS*, we trigger an immediate collection and resize the heap. This new polling mechanism allows us to adapt to bursts of memory pressure more quickly and effectively than our previous model.

4 VM System Design and Implementation

We now present the CRAMM VM system. We first describe why standard VM systems are insufficient for predictively adaptive heap sizing. We then describe the structure of the CRAMM VM system, followed by detailed discussions of how it calculates working set sizes and how it controls histogram collection overhead.

Given the heap sizing model presented in Section 3.1, the underlying VM system must provide to a GC-based process both its working set size (WSS) and its main memory allocation,² thus allowing the GC to choose a proper heap size. Unfortunately, we cannot easily obtain this information from standard VM systems, including the Linux VM.

Linux uses a global page replacement policy that manages each physical page within a single data structure for all processes and files. Linux thus has only ordinal information about all pages, giving each page a ranking among the total pool of pages. It has no cardinal information about the reference rates, nor any separation of pages according to process or file. Consequently, it cannot track the LRU reference histogram—the distribution of memory references to pages managed by an LRU queue-which is needed to determine the WSS for each process. Furthermore, it cannot predict how much it could reduce the allocations of files and other processes without inducing heavy page faulting. It therefore cannot wisely choose a main memory allocation to offer to a GC-based process. Finally, even if it chose to reduce the allocations for some files or other processes, global page replacement cannot guarantee that it will replace the pages of those processes first.

The CRAMM VM system addresses these limitations. Figure 2 gives an overview of the CRAMM VM structure and interface. For each file and process, it keeps separate page lists and an LRU reference histogram. It also tracks the mean cost of a major page fault (one that requires disk I/O) so that, along with the histogram and a desired maximum fault rate, it can compute the WSS of a process.

Its ability to compute the WSS of each file and process allows the CRAMM VM to calculate new allocations to each without causing thrashing by assigning too small an allocation. When an allocation is reduced, the separate page lists allow the VM to prefer reclaiming pages from those files and processes that are consuming more than their allocation.

A garbage collector communicates with the CRAMM VM through system calls. First, the collector registers itself as a cooperative process with the CRAMM VM at initialization time. The VM responds with the current amount of free memory, allowing the collector to pick a reasonable initial heap size. Second, after each heap collection, the collector requests a WSS estimate and a main memory allocation from the VM. The collector then uses this information to select a new heap size. If it changes its heap size, it calls on the VM to clear its old histogram, since the new

heap size will exhibit a different reference pattern.

Last, the collector periodically polls the VM for an estimate of the *free memory*—the main memory space that could be allocated to the process without causing others to thrash. If this value is unexpectedly low, then memory pressure has suddenly increased. Either some other system activity is aggressively consuming memory (e.g. the startup of a new process), or this process has more live data (increased *heap utilization*), and thus is using more memory than expected. The collector responds by pre-emptively collecting the heap and selecting a new heap size.

4.1 CRAMM VM Structure

The CRAMM VM allocates a data structure, called mem_info, for each *address space* (an inode for files or an mm_struct for processes). This structure comprises a list of pages, an LRU reference histogram, and some additional control fields.

Figure 4 shows the page list structure of a process. The CRAMM VM manages each address space (the space of a file or a process) much like the Linux VM manages its global queue. For the in-memory pages of each address space, it maintains a segmented queue (SEGQ) structure [5], where the active list contains the more recently used pages and the inactive list contains those less recently used. When a new page is faulted into memory, the VM places it at the head of the active list. If the addition of this page causes the active list to be too large, it moves pages from the tail of the active list to the head of the inactive list. When the process exceeds its main memory allocation, the VM removes a page from the tail of the inactive list and evicts it to disk. This page is then inserted at the head of a third segment, the evicted list. When an address space's WSS exceeds its main memory allocation, the evicted list's histogram data allows the VM to project how large the allocation must be to capture the working set.

The active list is managed using a CLOCK algorithm. The inactive list is ordered by each page's time of removal from the active list. The relative sizes of these two lists is controlled by an adaptive mechanism described in Section 4.3. Like a traditional SEGQ, all inactive pages have their access permissions removed, forcing any reference to an inactive page to cause a minor page fault. When such a page fault occurs, the VM restores the page's permissions and promotes it into the active list, and then updates the address space's histogram. The insertion of a new page into the active list may force other pages out of the active list. The VM manages the evicted list similarly; the only difference is that a reference to an evicted page triggers disk activity.

Replacement algorithm: The CRAMM VM places each mem_info structure into one of two lists: the *unused list* for the address spaces of files for which there are no open file descriptors, and the *normal list* for all other ad-

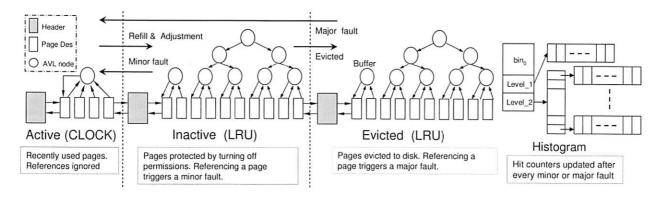


Figure 4: Segmented queue page lists for one address space (file or process).

dress spaces. When the VM must replace a page, it preferentially selects a mem_info from the unused list and then reclaims a page from the tail of that inactive list. If the unused list is empty, the VM selects a mem_info in a round robin manner from the normal list, and then selects a page from the tail of its inactive list.

As Section 5.2 shows, this eviction algorithm is less effective than the standard Linux VM replacement algorithm. However, the CRAMM VM structure can support standard replacement policies and algorithms while also presenting the possibility of new policies that control per-address-space main memory allocation explicitly.

Available Memory: A garbage collector will periodically request that the CRAMM VM report the available memory—the total main memory space that could be allocated to the process. Specifically, the CRAMM VM reports the available memory (available) as the sum of the process's resident set size (rss), the free main memory (free), and the total number of pages found in the unused list (unused). There is also space reserved by the VM (reserved) to maintain a minimal pool of free pages that must be subtracted from this sum:

$$available = rss + free + unused - reserved$$

This value is useful to the collector because the CRAMM VM's per-address-space structure allows it to allocate this much space to a process without causing any page swapping. Standard VM systems that use global memory management (e.g., Linux) cannot identify the unused file space or preclude the possibility of page swapping as memory is re-allocated to a process.

4.2 Working Set Size Calculation

The CRAMM VM tracks the current working set size of each process. Recall that the WSS is the smallest main memory allocation for which page faulting degrades process throughput by less than t%. If t = 0, space may be wasted by caching pages that receive very little use. When t is small but non-zero, the WSS may be substantially smaller than for t = 0, yet still yield only trivial page

swapping. In our experiments, we chose t = 5%.

In order to calculate the WSS, the VM maintains an LRU reference histogram h [25, 27] for each process. For each reference to a page at position i of the process's page lists, the VM increments h[i].³ This histogram allows the VM to calculate the number of page faults that would occur for each possible memory allocation. The VM also monitors the mean cost of a major fault (majfc) and the time T that each process has spent on the CPU. To calculate the WSS, it scans the histogram backward to find the allocation at which the number of page faults is just below $(T \times t)/majfc$.

Page list position: When a page fault occurs, the referenced page is found within the page lists using a hash map. In order to maintain the histograms, the CRAMM VM must determine the position of that page within the page lists. Because a linear traversal of the lists would be inefficient, the VM attaches an AVL tree to each page list. Figure 4 shows the structure that the VM uses to calculate page list positions in logarithmic time. Specifically, every leaf node in the AVL tree points to a linked list of up to k pages, where k depends on the list into which the node points. Every non-leaf node is annotated with the total number of pages in its subtree; additionally, each non-leaf node is assigned a capacity that is the k-values of its children. The VM puts newly added pages into a buffer, and inserts this buffer into the AVL tree as a leaf node when that buffer points to k pages. Whenever a non-leaf node drops to half full, the VM merges its children and adjusts the tree shape accordingly.

When a page is referenced, the VM first searches linearly to find the page's position in the containing leaf node. It then walks up the AVL tree, summing the pages in leaf nodes that point to earlier portions of the page list. Thus, given that *k* is constant and small, determining a page's list position is performed in time proportional to the height of the AVL tree.

Because the CRAMM VM does not track references to pages in the active list, one leaf node contains pointers to all pages in the active list, and for this leaf node, $k = \infty$. For leaf nodes that point to inactive and evicted pages, k = 64— a value chosen to balance the work of linear search and tree traversal. The AVL trees have low space overhead. Suppose an application has N 4KB pages, and our AVL node structure is 24 bytes long. Here, the worst case space overhead (all nodes half full, and the total number of nodes is twice the number of leaf nodes) is:

$$\frac{\left(\left(\frac{N}{64} \times 2 \times 2\right) \times 24 \ bytes\right)}{\left(N \times 4096 \ bytes\right)} < 0.037\%$$

On average, we observe that the active list contains a large portion (more than half) of the pages used by a process, and thus the overhead is even lower.

LRU histogram: Keeping one histogram entry for every page list position would incur a large space overhead. Instead, the CRAMM VM groups positions into bins. In our implementation, every bin corresponds to 64 pages (256 KB given the page size of 4 KB). This granularity is fine enough to provide a sufficiently accurate WSS measurement while reducing the space overhead substantially.

Furthermore, CRAMM dynamically allocates space for the histogram in chunks of 512 bytes. Given that a histogram entry is 8 bytes in size, one chunk corresponds to histogram entries for 16 MB of pages. Figure 4 shows the data structure for a histogram. We see that, when a process or file uses less than 64 pages (256 KB), it uses only bin₀. This approach optimizes space for the common case of small processes and files. Any process or file that requires more than 256 KB but less than 16MB memory uses the level_1 histogram. Larger ones use the level_2 histogram. The worst-case histogram space overhead occurs when a process uses exactly 65 pages. Here, the histogram will need about 0.2% of the memory consumed by the process. In common cases, it is about 8 bytes per 64 pages, which is less than 0.004%.

Major fault cost: Calculating WSS requires tracking the mean cost of a major page fault. The CRAMM VM keeps a single, system-wide estimate *majfc* of this cost. When the VM initiates a swap-in operation, it marks the page with a time-stamp. After the read completes, the VM calculates the time used to load the page. This new time is then used to update *majfc*.

4.3 Controlling Histogram Collection Overhead

Because the CRAMM VM updates a histogram entry at every reference to an inactive page, the size of the inactive list determines the overhead of histogram collection. If the inactive list is too large, then too much time will be spent handling minor page faults and updating histogram entries. If the inactive list is too small, then the histogram will provide too little information to calculate an accurate WSS. Thus, we want the inactive list to be as large as possible without inducing too much overhead.

The VM sets a target for *minor fault overhead*, expressed as a percentage increase in running time for processes, and dynamically adjusts the inactive list size according to this target. For each process, the VM tracks its CPU time T and a count of its minor page faults n. It also maintains a system-wide minor fault cost *minfc* using the same approach as with *majfc*. It uses these values to calculate the minor fault overhead as $(n \times minfc)/T$. It performs this calculation periodically, after which it resets both T and n. Given a target of 1% and a constant threshold for deviation from that target of 0.5%, one of three cases may apply:

- If the overhead exceeds 1.5%, the VM decreases the inactive list size.
- If the overhead is less than 0.5%, it increases the inactive list size.
- If there are no minor faults during this period, and if the inactive list is not full, then it moves pages from the active to the inactive list (refilling the inactive list).

This simple adaptive mechanism, set to a 1% overhead target and a 0.5% deviation threshold, successfully keeps the overhead low while yielding sufficient histogram information for WSS calculations.

Size adjustment calculations: CRAMM assigns each process a *target inactive size*, initially 0. When CRAMM adjusts the inactive list size, it is really setting this target size. Assume that a process has P_A pages in the active list and P_I in the inactive list. Depending on the overhead's relationship to its threshold, the new target will be:

- Increase: $P_I + max(min(P_A, P_I)/32, 8)$
- Decrease: $P_I max(min(P_A, P_I)/8, 8)$
- Refill: $P_I + max(min(min(P_A, P_I)/16, 256), 8)$

By choosing the smaller of P_A and P_I in these equations, we make the adjustments small if either list is small, thus not changing the target too drastically. These formulas also ensure that at least some constant change is applied to the target, ensuring a change that will have some effect. We also put an upper bound on the refilling adjustment to prevent flushing too many pages into the inactive list at a time. Finally, we decrease the target inactive list size more aggressively than we increase it because low overhead is a more critical and sensitive goal than accurate histogram information. We also refill more aggressively than we increase because an absence of minor faults is a strong indication of an inadequate inactive list size.

Whenever a page is added to the active list, the VM checks the current inactive list size. If it is less than its target, then the VM moves several pages from the active list to the inactive list (8 pages in our implementation). When an

adjustment triggers refilling, the VM immediately forces no more than 256 pages into the inactive list to match its new target. As this adjustment only resets the target size and usually does not move pages immediately, the algorithm is largely insensitive to the values of these parameters.

Adaptivity triggers: In the CRAMM VM, there are two events that can trigger an inactive list size adjustment. The first, adjust_interval, is based on running time, and the second, adjust_count, is based on the number of minor faults.

Every new process has its *adjust_interval* initialized to a default value ($\frac{1}{16}sec$). Whenever a process is scheduled, if its running time since the last adjustment exceeds its *adjust_interval* value, then the VM adjusts the inactive list size.

The adjust_count variable is initialized to be (adjust_interval × 2%)/minfc. If a process suffers this number of minor faults before adjust_interval CPU time has passed, then its overhead is well beyond the acceptable level. At each minor fault, the VM checks whether the number of minor faults since the last adjustment exceeds adjust_count. If so, it forces an adjustment.

5 Experimental Evaluation

We now evaluate our VM implementation and heap size manager. We first compare the performance of the CRAMM VM with the original Linux VM. We then add the heap size manager to several collectors in Jikes RVM, and evaluate their performance under both static and dynamic real memory allocations. We also compare them with the JRockit [7] and HotSpot [19] JVMs under similar conditions. Finally, we run two concurrent instances of our adaptive collectors under memory pressure to see how they interact with each other.

5.1 Methodology Overview

We perform all measurements on a 1.70GHz Pentium 4 Linux machine with 512MB of RAM and 512MB of local swap space. The processor has 12KB I and 8KB D L1 caches and a 256KB unified L2 cache. We installed both the "stock" Linux kernel (version 2.4.20) and our CRAMM kernel. We run each of our experiments six times in single-user mode, and report the mean of the last five runs. In order to simulate memory pressure, we use a background process to pin a certain volume of pages in memory using mlock.

Application platform: We used Jikes RVM v2.4.1 [3] built for Linux x86 as our Java platform. We optimized the system images to the highest optimization level to avoid run-time compilation of those components. Jikes RVM uses an *adaptive* compilation system, which invokes optimization based on time-driven sampling. This makes executions non-deterministic. In order to get comparable deterministic executions, we took compilation logs from 7 runs of each benchmark using the adaptive system, and di-

rected the system to compile methods according to the log from the run with the best performance. This is called the *replay* system. It is deterministic and highly similar to typical adaptive system runs.

Collectors: We evaluate five collectors from the MMTk memory management toolkit [9] in Jikes RVM: MS (marksweep), GenMS (generational mark-sweep), CopyMS (copying mark-sweep), SS (semi-space), and GenCopy (generational copying). All of these collectors have a separate non-copying region for large objects (2KB or more). collected with the Treadmill algorithm [6]. They also use separate non-copying regions for meta-data and immortal objects. We now describe the other regions each collector uses for ordinary small objects. MS is non-generational with a single MS region. GenMS is generational with a copying nursery and MS mature space. CopyMS is nongenerational with two regions, both collected at every GC. New objects go into a copy region, while copy survivors go into an MS region. SS is non-generational with a single copying region. GenCopy is generational with copying nursery and mature space. Both generational collectors (GenMS and GenCopy) use Appel-style nursery sizing [4] (starts large and shrinks as mature space grows).

Benchmarks: For evaluating JVM performance, we ran all benchmarks from the SPECjvm98 suite (standard and widely used), plus those benchmarks from the Da-Capo suite [10] (an emerging standard for JVM GC evaluation) that run under Jikes RVM, plus ipsixql (a publicly available XML database program) and pseudojbb (a variant of the standard, often-used SPECjbb server benchmark with a fixed workload (140,000 transactions) instead of fixed time limit). For evaluating general VM performance, we used the standard SPEC2000 suite.

Presented: Many results are similar, so to save space we present results only from some representative collectors and benchmarks. For collectors, we chose SS, MS, and GenMS to cover copying, non-copying, and generational variants. For benchmarks, we chose <code>javac</code>, <code>jack</code>, <code>pseudojbb</code>, <code>ipsixql</code>, <code>jython</code>, and <code>pmd</code>.

5.2 VM Performance

For the CRAMM VM to be practical, its baseline performance (i.e., while collecting useful histogram/working set size information) must be competitive when physical RAM is plentiful. We compare the performance of the CRAMM VM to that of the stock Linux kernel across our entire benchmark suite.⁴ For each benchmark, we use the input that makes it runs longer than 60 seconds.

Figure 5 summarizes the results, which are geometric means across all benchmarks: SPEC2000int, SPEC2000fp, and all the Java benchmarks (SPECjvm98, DaCapo, pseudojbb, and ipsixql) with five different garbage collectors. While the inactive list size adjustment mechanism effectively keeps the cost of collecting histogram data in the

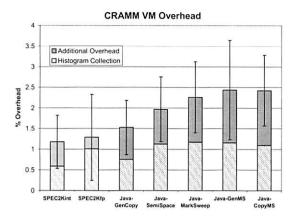


Figure 5: Virtual memory overhead (% increase in execution time) without paging, across all benchmark suites and garbage collectors.

desired range (e.g., 0.59% for SPEC2Kint and 1.02% for SPEC2Kfp), the slowdown is generally about 1–2.5%. We believe this overhead is caused by CRAMM polluting the cache when handling minor faults as it processes page lists and AVL trees. This, in turn, leads to extra cache misses for the application. We verified that at the target minor fault overhead, CRAMM incurs enough minor faults to calculate the working set size accurately with respect to our 5% page fault threshold.

CRAMM's performance is generally somewhat poorer on the Java benchmarks, where it must spend more time handling minor faults caused by the dramatic working set changes between the mutator and collector phases of GCed applications. However, the fault handling overhead remains in our target range. Overall, CRAMM collects the necessary information at very low overhead in most cases, and its performance is competitive with that of the stock kernel.

5.3 Static Memory Allocation

To test our adaptive mechanism, we run the benchmarks over a range of requested heap sizes with a fixed memory allocation. We select memory allocations that reveal the effects of large heaps in small allocations and small heaps in large allocations. In particular, we try to evaluate the ability of our mechanism to grow and shrink the heap. We run the non-adaptive collectors (which simply use the requested heap size) on both the stock and CRAMM kernels, and the adaptive collectors on the CRAMM kernel, and compare performance.

Figure 6 and Figure 7 show execution time for benchmarks using MS and GenMS collectors, respectively, with a static memory allocation. For almost every combination of benchmark and requested heap size, our adaptive collector chooses a heap size that is nearly optimal. It either reduces total execution time dramatically, or performs at least as well as the non-adaptive collector. At the leftmost

side of each curve, the non-adaptive collector runs at a heap size that does not consume the entire allocation, thus under-utilizing available memory, collecting too frequently and inducing high GC overhead. The adaptive collector grows the heap size to reduce the number of collections without incurring paging. At the smallest requested heap sizes, this adjustment reduces execution time by as much as 85%.

At slightly larger requested heap sizes, the non-adaptive collector performs fewer collections, better utilizing available memory. One can see that there is an ideal heap size for the given benchmark and allocation. At that heap size, the non-adaptive collector performs well—but the adaptive collector often matches it, and is never much worse. The maximum slowdown we observed is 11% across all the benchmarks. Our working set size calculation uses a page fault threshold of t=5%, so we are allowing a trivial amount of paging—while reducing the working set size substantially.

Once the requested heap size goes slightly beyond the ideal, non-adaptive collector performance drops dramatically. The working set size is just slightly too large for the allocation, which induces enough paging to slow execution by as much as a factor of 5 to 10. In contrast, our adaptive collector shrinks the heap so that the allocation completely captures the working set size. By performing slightly more frequent collections, the adaptive collector consumes a modest amount of CPU time to avoid a lot of paging, thus reducing elapsed time by as much as 90%. When the requested heap size becomes even larger, the performance of our adaptive collector remains the same. However, the execution time of the non-adaptive collector decreases gradually. This is because it does fewer collections, and it is collections that cause most of the paging.

Interestingly, when we disable adaptivity, the CRAMM VM exhibits worse paging performance than the stock Linux VM. LRU-based eviction algorithm turns out to be a poor fit for garbage collection's memory reference behavior. Collectors typically exhibit loop-like behavior when tracing live objects, and LRU is notoriously bad in handling large loops. The Linux VM instead uses an eviction algorithm based on a combination of CLOCK and a linear scan over the program's address space, which happens to work better in this case.

5.4 Dynamic Memory Allocation

The results given so far show that our adaptive mechanism selects a good heap size when presented with an unchanging memory allocation. We now examine how CRAMM performs when the memory allocation changes dynamically. To simulate dynamic memory pressure, we use a background process that repeatedly consumes and releases memory. Specifically, it consists of an infinite loop, in which each iteration first sleeps for 25 seconds, then mmap's 50MB memory, mlock's it for 50 seconds, and

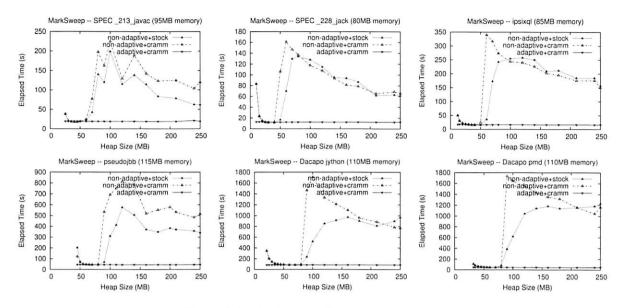


Figure 6: Static Memory Allocation: MarkSweep

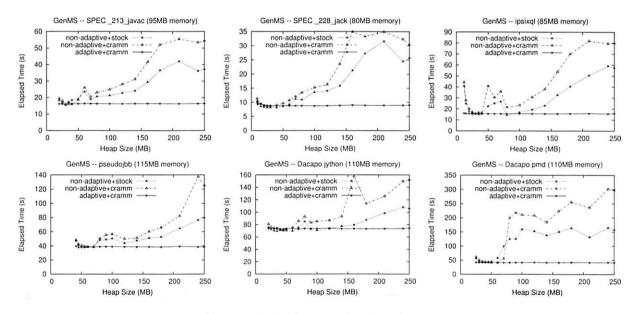


Figure 7: Static Memory Allocation: GenMS

then unlocks and unmaps the memory. We also modify how we invoke benchmarks so that they run long enough to measure: we give pseudojbb a large transaction number, and iterate javac 20 times.

Table 1 summarizes the performance of both the non-adaptive and adaptive collectors under this dynamic memory pressure. The first column gives the benchmarks and their initial memory allocation. The second column gives the collectors and their requested heap sizes respectively. We set the requested heap size so that the benchmark will run gracefully in the initial memory allocation. We present the total elapsed time (T), CPU utilization (cpu), and num-

ber of major faults (MF) for each collector. We compare them against running the benchmark at the requested heap size with sufficient memory. The last column shows adaptive execution time relative to non-adaptive. We see that, for each collector, the adaptive mechanism adjusts the heap size in response to memory pressure, nearly eliminating paging. The adaptive collectors show high CPU utilization and dramatically reduced execution times.

Figure 8 illustrates how our adaptive collectors change the heap size while running pseudojbb under dynamic memory pressure. The graphs in the first row demonstrate how available memory changes over time, and shows the

Benchmark	Collector (Heap Size)		Enough Memory		Adaptive Collector			Non-Adaptive Collector			Adaptive
(Memory)			T(sec) M	MF	T(sec)	cpu	MF	T(sec)	cpu	MF	Yes/No
pseudojbb	SS	(160M)	297.35	1136	339.91	99%	1451	501.62	65%	24382	0.678
(160M)	MS	(120M)	336.17	1136	386.88	98%	1179	928.49	36%	47941	0.417
	GenMS	(120M)	296.67	1136	302.53	98%	1613	720.11	48%	39944	0.420
javac	SS	(150M)	237.51	1129	259.35	94%	1596	455.38	68%	24047	0.569
(140M)	MS	(90M)	261.63	1129	288.09	95%	1789	555.92	47%	25954	0.518
	GenMS	(90M)	249.02	1129	263.69	95%	2073	541.87	50%	33712	0.487

Table 1: Dynamic Memory Allocation: Performance of Adaptive vs. Non-Adaptive Collectors

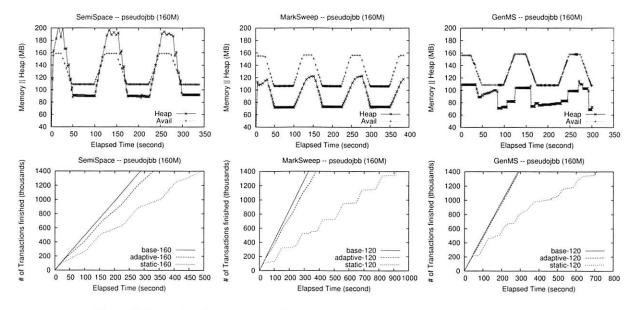


Figure 8: Dynamic Memory Allocation (pseudojbb): Heap Adjustment and Throughput

corresponding heap size chosen by each adaptive collector. We see that as available memory drops, the adaptive collectors quickly shrink the heap to avoid paging. However, if our adaptive collectors do not periodically poll for current available memory and collect before the heap is filled up, the CPU utilization falls below 80% and can be as low as 53%. Likewise, the adaptive collectors grow the heap when there is more available memory. One can also see that the difference between the maximum and minimum heap size is approximately the amount of memory change divided by heap utilization u, consistent with our working set size model presented in Section 3.1.

We also compare the throughput of the adaptive and non-adaptive collectors (the second row in Figure 8), by printing out the number of transactions finished as time elapses for pseudojbb. These curves show that memory pressure has much less impact on throughput when running under our adaptive collectors. It causes only a small disturbance and only for a short period of time. The total execution time of our adaptive collectors is a little longer than that of the base case, simply because they run at a much smaller heap size (and thus collect more often) when there is less

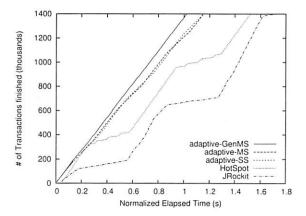


Figure 9: Throughput under dynamic memory pressure, versus JRockit and HotSpot.

memory. The non-adaptive collectors experience significant paging slowdown when under memory pressure.

As previously mentioned, JRockit and HotSpot do not adjust heap size well in response to changing memory allocation. Figure 9 compares the throughput of our adaptive collectors with that of JRockit and HotSpot. We care-

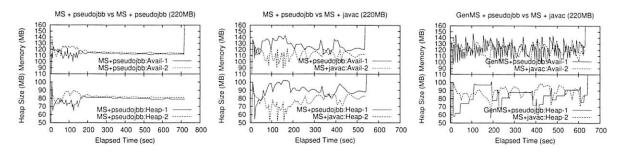


Figure 10: Running Two Instances of Adaptive Collectors.

fully choose the initial memory allocation so that the background process imposes the same amount of relative memory pressure as for our adaptive collectors. However, being an experimental platform, Jikes RVM's compiler does not produce as efficient code as these commercial JVMs. We thus normalize the time for each of them to the total execution time that each JVM takes to run when given ample physical memory. The results show that both JRockit and HotSpot experience a large relative performance loss. The flat regions on their throughput curves indicate that they make barely any progress when available memory suddenly shrinks to less than their working set. Meanwhile, our adaptive collector changes the heap size to fit in available memory, maintaining high performance.

Finally, we examine how our adaptive collectors interact with each other. We start two instances using adaptive collectors with a certain memory allocation (220MB), and let them adjust their heap sizes independently. We explore several combinations of collector and benchmark: the same collector and benchmark, the same collector and different benchmarks, and different collectors with different benchmarks. The experiments show that, for all these combinations, our adaptive collectors keep CPU utilization at least 91%. Figure 10 shows the amount of available memory observed by each collector and their adapted heap size over time. We see that, after bouncing around a little, our adaptive collectors tend to converge to heap sizes that give each job a fair share of available memory, even though each works independently. The curves of GenMS in the third graph show how filtering out small nursery collections helps to stabilize heap size. This experiment still focuses on how each JVM adapts to dynamic memory allocation. Although our mechanism effectively prevents each of them from paging, the memory allocation to each JVM may be unfair. We leave to future work how best to divide memory among multiple competing JVMs.

6 Related Work

We now discuss the work most closely related to CRAMM, first discussing work related to the CRAMM VM and then addressing GC-based approaches to sizing the heap.

6.1 Virtual Memory

The CRAMM VM computes stack distances, which were originally designed for trace analysis. Mattson et al. introduced a one-pass algorithm, based on stack distances, that analyzes a reference trace and produces cache misses for caches of any size [22]. This algorithm was later adapted by Kim and Hsu to handle highly-associative caches [21]. However, these algorithms compute a stack distance in linear time, making them too slow to use inside a kernel. Subsequent work on analyzing reference traces used more advanced dictionary data structures [1, 8, 17, 23, 26]. These algorithms calculate a stack distance in logarithmic time, but do not maintain underlying referenced blocks in order. This order is unnecessary for trace processing but crucial for page eviction decisions. The CRAMM VM maintains pages in a list that preserves potential eviction order, and uses a separate AVL tree to calculate a stack distance in logarithmic time.

Zhou et al. present a VM system that also tracks LRU reference curves inside the kernel [29]. They use Kim and Hsu's linear-time algorithm to maintain LRU order and calculate stack distances. To achieve reasonable efficiency, this algorithm requires the use of large group sizes (e.g., 1024 pages) that significantly degrade accuracy. They also use a static division between the active and inactive lists, yielding an overhead of 7 to 10%. The CRAMM VM computes the stack distance in logarithmic time, and can track reference histograms at arbitrary granularities. Furthermore, its inactive list size adjustment algorithm allows it to collect information accurately from the tail of miss curves while limiting reference histogram overhead to 1%.

6.2 Garbage Collection

Researchers have proposed a number of heap sizing approaches for garbage collection; Table 2 provides a summary. The closest work to CRAMM is by Alonso and Appel, who also exploit VM system information to adjust the heap size [2]. Their collector periodically queries the VM for the current amount of available memory and adjusts the heap size in response. CRAMM differs from this work in several key respects. While their approach shrinks the heap when memory pressure is high, it does not expand

	Grows	Shrinks	Static	Dynamic	Collector	Needs OS	
	Heap	Heap	Allocation	Allocation	Neutral	Support	Responds to
Alonso et al.[2]		\checkmark	\checkmark	√		√	memory allocation
Brecht et al.[11]	\vee		✓				pre-defined rules
Cooper et al.[14]	\vee		√				user supplied target
BC [18]		\checkmark	✓	√		\checkmark	page swapping
JRockit [7]	\vee	\checkmark	\checkmark		√		throughput or pause time
HotSpot [19]	\vee	\checkmark	✓				throughput and pause time
MMTk [9]	\vee	\vee	✓		√		live ratio and GC load
CRAMM/AHS [28]	\vee	\checkmark	✓	\checkmark	\vee	✓	memory allocation

Table 2: A comparison of approaches to dynamic heap sizing.

and thus reduce GC frequency when pressure is low. It also relies on standard interfaces to the VM system that provide a coarse and often inaccurate estimate of memory pressure. The CRAMM VM captures detailed reference information and provides reliable values.

Brecht et al. adapt Alonso and Appel's approach to control heap growth via ad hoc rules for two given *static* memory sizes [11]. Cooper et al. dynamically adjust the heap size of an Appel-style collector according to a *user-supplied* memory usage target [14]. If the target matches the amount of free memory, their approach adjusts the heap to make full use of it. However, none of these approaches can adjust to dynamic memory allocations. CRAMM automatically identifies an optimal heap size using data from the VM. Furthermore, the CRAMM model captures the relationship between working set size and heap size, making its approach more general and robust.

Our research group previously presented the bookmarking collector (BC), a garbage-collection algorithm that guides a lightly modified VM system to evict pages that do not contain live objects and installs "bookmarks" in pages in response to eviction notifications [18]. These bookmarks allow BC to collect the heap without touching already evicted pages, which CRAMM must. One shortcoming of BC is that it currently cannot grow the heap because it responds only to page eviction notifications, while CRAMM both shrinks and grows the heap to fit. We view BC as orthogonal and complementary to the work presented here.

7 Conclusion

We present CRAMM, a new system designed to support garbage-collected applications. CRAMM combines a new virtual memory system with a garbage-collectorneutral, analytic heap sizing model to dynamically adjust heap sizes. In exchange for modest overhead (1–2.5% on average), CRAMM can improve performance dramatically. CRAMM allows garbage-collected applications to run with a nearly-optimal heap size in the absence of memory pressure, and adapts quickly to dynamic memory pressure changes, avoiding paging while providing high CPU utilization.

8 Acknowledgements

This material is based upon work supported by the National Science Foundation under grant number CCR-0085792 and CAREER Award CNS-0347339. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

- G. Almasi, C. Cascaval, and D. A. Padua. Calculating stack distances efficiently. In ACM SIGPLAN Workshop on Memory System Performance, pages 37–43, Berlin, Germany, Oct. 2002.
- [2] R. Alonso and A. W. Appel. An advisor for flexible working sets. In *Proceedings of the 1990 SIGMETRICS* Conference on Measurement and Modeling of Computer Systems, pages 153–162, Boulder, CO, May 1990.
- [3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalepeño virtual machine. *IBM Systems Journal*, 39(1):211–238, Feb. 2000.
- [4] A. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, Feb. 1989.
- [5] O. Babaoglu and D. Ferrari. Two-level replacement decisions in paging stores. *IEEE Transactions on Computers*, C-32(12):1151–1159, Dec. 1983.
- [6] H. G. Baker. The Treadmill: Real-time garbage collection without motion sickness. ACM SIGPLAN Notices, 27(3):66–70, March 1992.
- [7] BEA WebLogic. Technical white paper JRockit: Java for the enterprise. http://www.bea.com/content/news_events /white_papers/BEA_JRockit_wp.pdf.
- [8] B. T. Bennett and V. J. Kruskal. LRU stack processing. IBM Journal of R & D, 19(4):353–357, 1975.
- [9] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In 26th International Conference on Software Engineering, pages 137–146, May 2004.

- [10] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, New York, NY, USA, Oct. 2006. ACM Press.
- [11] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *Proceedings of the 2001 ACM* SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications, pages 353–366, Tampa, FL, June 2001.
- [12] R. W. Carr and J. L. Henessey. WSClock a simple and effective algorithm for virtual memory management. In Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP), pages 87–95, Dec. 1981.
- [13] W. W. Chu and H. Opderbeck. The page fault frequency replacement algorithm. In AFIPS Conference Proceedings, volume 41(1), pages 597–609, Montvale, NJ, 1972. AFIPS Press.
- [14] E. Cooper, S. Nettles, and I. Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming, pages 43–52, San Francisco, CA, June 1992.
- [15] P. J. Denning. The working set model for program behavior. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 15.1–15.12, Jan. 1967.
- [16] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, Jan. 1980.
- [17] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pages 245–257, San Diego, CA, June 2003.
- [18] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementaton*, pages 143–153, Chicago, IL, June 2005.
- [19] JavaSoft. J2SE 1.5.0 documentation: Garbage collector ergonomics. http://java.sun.com/j2se/1.5.0/docs/guide/vm/ gc-ergonomics.html.
- [20] S. F. Kaplan, L. A. McGeoch, and M. F. Cole. Adaptive caching for demand prepaging. In *Proceedings of the 2002 International Symposium on Memory Management*, pages 114–126, June 2002.
- [21] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *Proceedings of the 1991 SIGMETRICS Conference on*

- Measurement and Modeling of Computer Systems, pages 212–213, San Diego, CA, 1991.
- [22] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.
- [23] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981.
- [24] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, pages 79–95, New York, NY, USA, 1995. ACM Press.
- [25] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. The EELRU adaptive replacement algorithm. *Performance Evaluation*, 53(2):93–123, July 2003.
- [26] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Measurement and Modeling of Computer Systems*, pages 24–35, Santa Clara, CA, 1993.
- [27] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of The 1999 USENIX Annual Technical Conference*, pages 101–116, Monterey, California, June 1999. USENIX Association.
- [28] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, pages 61–72, Vancouver, Canada, Oct. 2004.
- [29] P. Zhou, V. Pandy, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curves for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–188, Boston, MA, Oct. 2004.

Notes

We use VM to denote virtual memory throughout this paper.

²The *main memory allocation* is not the same as the *resident set size*. The latter is the amount of main memory currently consumed by a process, while the former is the amount of main memory that the VM system is willing to let the process consume before evicting its pages.

³Notice that we refer to the histogram as an *LRU reference histogram*, but that our page lists are not in true LRU order, and so the histogram is really a *SegQ reference histogram*. Also, note that only references to the inactive and evicted lists are applicable here, since references to active pages occur without kernel intervention.

⁴We could not compile and run some SPEC2000 Fortran programs, so we omit some of the FP benchmarks.

Flight Data Recorder: Monitoring Persistent-State Interactions to Improve Systems Management

Chad Verbowski, Emre Kıcıman, Arunvijay Kumar, Brad Daniels, Shan Lu[‡], Juhan Lee^{*}, Yi-Min Wang, Roussi Roussev[†] Microsoft Research, [†]Florida Institute of Technology, [‡]U. of Illinois at Urbana-Champaign, ^{*}Microsoft MSN

Abstract

Mismanagement of the persistent state of a system—all the executable files, configuration settings and other data that govern how a system functions—causes reliability problems, security vulnerabilities, and drives up operation costs. Recent research traces persistent state interactions—how state is read, modified, etc.—to help troubleshooting, change management and malware mitigation, but has been limited by the difficulty of collecting, storing, and analyzing the 10s to 100s of millions of daily events that occur on a single machine, much less the 1000s or more machines in many computing environments.

We present the Flight Data Recorder (FDR) that enables *always-on tracing, storage and analysis* of persistent state interactions. FDR uses a domain-specific log format, tailored to observed file system workloads and common systems management queries. Our lossless log format compresses logs to only 0.5-0.9 bytes per interaction. In this log format, 1000 machine-days of logs—over 25 billion events—can be analyzed in less than 30 minutes. We report on our deployment of FDR to 207 production machines at MSN, and show that a single centralized collection machine can potentially scale to collecting and analyzing the complete records of persistent state interactions from 4000+ machines. Furthermore, our tracing technology is shipping as part of the Windows Vista OS.

1. Introduction

Misconfigurations and other persistent state (PS) problems are among the primary causes of failures and security vulnerabilities across a wide variety of systems, from individual desktop machines to largescale Internet services. MSN, a large Internet service, finds that, in one of their services running a 7000 machine system, 70% of problems not solved by rebooting were related to PS corruptions, while only 30% were hardware failures. In [24], Oppenheimer et al. find that configuration errors are the largest category of operator mistakes that lead to downtime in Internet services. Studies of wide-area networks show that misconfigurations cause 3 out of 4 BGP routing announcements, and are also a significant cause of extra load on DNS root servers [4,22]. Our own analysis of call logs from a large software company's internal help desk, responsible for managing corporate desktops, found that a plurality of their calls (28%) were PS related.1 Furthermore, most reported security compromises are against known vulnerabilities—administrators are wary of patching their systems because they do not know the state of their systems and cannot predict the impact of a change [1,26,34].

PS management is the process of maintaining the "correctness" of critical program files and settings to avoid the misconfigurations and inconsistencies that

There are three desired attributes in a tracing and analysis infrastructure. First is low performance overhead on the monitored client, such that it is feasible to always be collecting complete information for use by systems management tools. The second desired attribute is an efficient method to store data, so that we can collect logs from many machines over an extended period to provide a breadth and historical depth of data when managing systems. Finally, the analysis of these large volumes of data has to be scalable, so that we can monitor, analyze and manage today's large computing environments. Unfortunately, while many tracers have provided low-overhead, none of the state-of-the-art technologies for "always-on" tracing of PS interactions provide for efficient storage and analysis.

We present the Flight-Data Recorder (FDR), a highperformance, always-on tracer that provides complete records of PS interactions. Our primary contribution is a domain-specific, queryable and compressed log file

cause these reliability and security problems. Recent work has shown that *selectively* logging how processes running on a system interact with PS (e.g., read, write, create, delete) can be an important tool for quickly troubleshooting configuration problems, managing the impact of software patches, analyzing hacker break-ins, and detecting malicious websites exploiting web browsers [17,35-37]. Unfortunately, each of these techniques is limited by the current infeasibility of collecting and analyzing the complete logs of 10s to 100s of millions of events generated by a single machine, much less the 1000s of machines in even a medium-sized computing and IT environments.

¹ The other calls were related to hardware problems (17%), software bugs (15%), design problems (6%), "how to" calls (9%) and unclassified calls (12%). 19% not classified.

format, designed to exploit workload characteristics of PS interactions and key aspects of common-case queries—primarily that most systems management tasks are looking for "the needle in the haystack," searching for a small subset of PS interactions that meet well-defined criteria. The result is a highly efficient log format, requiring only 0.47-0.91 bytes per interaction, that supports the analysis of 1000 machine-days of logs, over 25 billion events, in less than 30 minutes.

We evaluate FDR's performance overhead, compression rates, query performance, and scalability. We also report our experiences with a deployment of FDR to monitor 207 production servers at various MSN sites. We describe how *always-on* tracing and analysis improve our ability to do after-the-fact queries on hard-to-reproduce incidents, provide insight into on-going system behaviors, and help administrators scalably manage large-scale systems such as IT environments and Internet service clusters.

In the next section, we discuss related work and the strengths and weaknesses of current approaches to tracing systems. We present FDR's architecture and log format design in sections 3 and 4, and evaluate the system in Section 5. Section 6 presents several analysis techniques that show how PS interactions can help systems management tasks like troubleshooting and change management. In Section 7, we discuss the implications of this work, and then conclude.

Throughout the paper, we use the term PS *entries* to refer to files and folders within the file system, as well as their equivalents within structured files such as the Windows Registry. A PS *interaction* is any kind of access, such as an open, read, write, close or delete operation.

2. Related Work

In this section, we discuss related research and common tools for tracing system behaviors. We discuss related work on analyzing and applying these traces to solve systems problems in Section 6. Table 1 compares the log-sizes and performance overhead of FDR and other systems described in this section for which we had data available [33,11,21,20,40].

The tools closest in mechanics to FDR are file system workload tracers. While, to our knowledge, FDR is the first attempt to analyze PS interactions to improve systems management, many past efforts have analyzed file system workload traces with the goal of optimizing disk layout, replication, etc. to improve I/O system performance [3,9,12,15,25,28,29,33]. Tracers based on some form of kernel instrumentation, like FDR and DTrace [30], can record complete information. While some tracers have had reasonable performance overheads, their main limitation has been a lack of support for efficient queries and the large log sizes. Tracers based on sniffing network file system traffic,

Table 1: Performance overhead and log sizes for related tracers. VTrace, Vogel and RFS track similar information to FDR. ReVirt and Forensix track more detailed information. Only FDR and Forensix provide explicit query support for traces.

	Performance	Log size	Log Size (MB/machine-day)		
	Overhead	(B/event)			
FDR	<1%	0.7	20MB		
VTrace	5-13%	3-20	N/A		
Vogel	0.5%	105	N/A		
RFS	<6%	N/A	709MB		
ReVirt	0-70%	N/A	40MB-1.4GB		
Forensix	6-37%	N/A	450MB		

such as NFS workload tracers [12,29] avoid any clientperceived performance penalties, but sacrifice visibility into requests satisfied by local caches as well as visibility of the process making a request.

Complete versioning file systems, such as CVFS [31] and WayBack [8] record separate versions of files for every write to the file system. While such file systems have been used as a tool in configuration debugging [39], they do not capture file reads, or details of the processes and users that are changing files. The Repairable File Service (RFS) logs file versioning information and also tracks information-flow through files and processes to analyze system intrusions [40].

In [33], Vogels declares analysis of his 190M trace records to be a "significant problem," and uses data warehousing techniques to analyze his data. The Forensix project, tracing system calls, also records logs in a standard database to achieve queryability [13]. However, Forensix's client-side performance overhead and their query performance (analyzing 7 machine-days of logs in 8-11 minutes) make it an unattractive option for large-scale production environments.

A very different approach to tracing a system's behavior is to record the nondeterministic events that affect the system, and combine this trace with virtual machine-based replay support. While this provides finer-grained and more detailed information about all the behaviors of a system than does FDR, this extra information can come at a high cost: ReVirt reports workload-dependent slowdowns up to 70% [11]. More significantly, arbitrary queries are not supported without replaying the execution of the virtual machine, taking time proportional to its original execution.

While, to our knowledge, we are the first to investigate domain-specific compression techniques for PS interaction or file system workload traces, there has been related work in the area on optimizing or compressing program CPU instruction traces [5,19], as

well as work to support data compression within general-purpose databases [6].

3. Flight Data Recorder Architecture

In this section, we present our architecture and implementation for black-box monitoring, collecting, and analysis of PS interactions. Our architecture consists of (1) a low-level driver that intercepts all PS interactions with the file system and the Windows Registry, calls to the APIs for process creation and binary load activity, and exposes an extensibility API for receiving PS interaction events from other specialized stores; and (2) a user mode daemon that collects and compresses the trace events into log files and uploads them to a central server, (3) a central server that aggregates the log files and, (4) an extensible set of query tools for analyzing the data stream. implementation does not require any changes to the core operating system or applications running atop it. We provide detailed discussion of our domain-specific queryable log format in Section 4.

3.1 FDR Agent Kernel-Mode Driver

Our low-level instrumentation is handled by a kernel mode boot driver², which operates in real-time and, for each PS interaction, records the current timestamp, process ID, thread ID, user ID, interaction type (read, write, etc.), and hashes of data values where applicable. For accesses to the file system, the driver records the path and filename, whether the access is to a file or a directory and, if applicable, the number of bytes read or written. For accesses to the registry, the driver records the name and location of the registry entry as well as the data it contains. The driver sits above the file system cache, but below the memory mapping manager. This driver also records process tree information, noting when a binary module is loaded, or when a process spawns another.

The largest performance impact from the driver stems from I/O related to log writing, memory copies related to logging events, and latency introduced by doing this work on the calling application's thread. We mitigate this by only using the application's thread to write the relevant records directly into the user-mode daemon's memory space, and doing the processing on the user-mode daemon's thread. Caches for user names and file names that need to be resolved for each interaction also help to minimize lookup costs.

Our kernel driver is stable and suitable for use in production environments, and will be available for public use as part of Windows Vista.

3.2 FDR Agent User-Mode Daemon

The user-mode daemon is responsible for receiving records of PS interactions from the kernel driver,

compressing them into our log format in-memory, and periodically uploading these logs to a central server.

To avoid impacting the performance of the system, we configure our daemon to run at lowest-priority, meaning it will be scheduled only if the CPU is otherwise idle. If the daemon does fall behind, the driver can be configured to either block until space is available or drop the event. However, in practice, we have found that a 4MB buffer is sufficient to avoid any loss on even our busiest server machines.

The daemon throttles its overall memory usage by monitoring the in-memory compressed log size, and flushing this to disk when it reaches a configurable threshold (typically 20MB to 50MB). The daemon will also periodically flush logs to disk to ensure reliable log collection in the event of agent or system failure. These logs are uploaded to a central server using a standard SMB network file system protocol. If a failure occurs during upload the daemon will save the log locally and periodically retry the upload.

The daemon also manages its own operation, for example, by automatically update its binaries and configuration settings when indicated on the central server, and monitoring its disk space and memory usage. Setting up FDR tracing on a new machine is simple: a user only needs to run a single binary on the machine and configure the log upload location.

3.3 FDR Collection Server

The collection server is responsible for organizing FDR log files as they are uploaded, triggering relevant query tools to analyze the files as they arrive, and pruning old log files from the archive. It also sets the appropriate access privileges and security on the collected files and processed data.

3.4 FDR Query Tools

The final pieces of our framework are the query tools that analyze log files as they arrive. Each query tool is specialized to answer a specific type of query for a systems management task. Simple example queries include "what files were modified today?", or "which programs depend on this configuration setting?" As all our log files are read-only, we do not require complicated transactional semantics or other coordination between our query tools. Each query tool reads the log files it is interested in scanning and implements its own query plan against the data within. While future work might investigate benefits of caching, sharing intermediate results across multiple concurrent queries, or other optimization techniques from the database literature, we found that allowing uncoordinated reads simplified the process of building new query tools as required.

4. Designing the Log Format

The key requirements we have for FDR's log format are that 1) logs are compact, so that their size does not

A kernel-mode boot driver is the first code to run after booting and the last to stop if the system is shut down.

overly burden client resources, network bandwidth or server-side scalability; and 2) the log format efficiently supports common-case queries. To meet these requirements, we built a preliminary version of FDR with a straightforward, flat format, and collected 5000 machine-days of traces from a wide variety of machines. We can personally attest to the difficulty of collecting, storing and analyzing this scale of data without support for compression and queryability. Based on our analysis of these traces, and a survey of how previous work applies such traces to systems management tasks, we designed an optimized log file format that takes advantage of three aspects of PS interaction workloads that we saw across our collected traces.

First, most PS interactions repeat many times during a day—93-99% of daily activity is a duplicate of an earlier event. For queries that care only about what happened, rather than when or how often, we can improve query performance by separating the definitions of this small number of distinct interactions from the details of when they occur.

Secondly, we observe that PS interactions are highly bursty, with many interactions occurring almost simultaneously and long idle periods between bursts. This allows us to save significant storage space by amortizing timestamp information across a burst.

Finally, we find that sequences of PS interactions are also highly repetitious; if we see a sequence of PS reads and writes, we are very likely to see the same sequence again in the future. This leads us to apply standard compression schemes to the time-ordered traces of PS interactions, achieving a high compression rate.

In the rest of this section, we describe relevant attributes of common-case queries, present the results and implications of our survey of PS interaction traces, and then describe the details of our log format.

4.1 Common Queries

Today, systems administrators deal with large-scale, complicated systems. According to surveys [9,28,33,36], an average Windows machine has 70k files and 200k registry settings. Faced with the task of managing these systems, a systems administrator's job is often a problem of "finding the needle in the haystack." For example, troubleshooting is the task of finding the few configuration settings or program files that are causing a problem; and to test a software upgrade or patch, the administrator needs to know what subset of the system might be affected by the change. To be useful, FDR must help systems administrators quickly identify the small set of relevant state and events out of all the state existing and events occurring across the many machines of a computing or IT environment. We describe the details of how systems management tasks use PS interaction traces in Section 6. Here, we briefly describe the aspects of

common-case queries that informed our log format design.

We find that most common systems management queries of PS interaction traces search for a subset of events, identified by the responsible process or user, the file or registry entry being accessed, or another aspect of the interaction ("Who changed this configuration?" or "What did I change yesterday?"). This means that, by organizing or indexing our log format around such attributes, we can quickly identify the subset of interactions of interest. Common queries are also often restricted by time range, looking only at events that occurred during a specific period, implying that our logs should support random access over time, not just sequential access.

Many systems management tasks only involve the existence (or absence) of a particular PS interaction, and not when or how often the interaction occurred. For example, finding all loads of a shared library, regardless of when they occurred, can identify the processes that depend on that library and help assess the impact of a software upgrade. Other times, queries do care about when a PS interaction occurred, but only need to know an interaction's relative-ordering vis-à-vis other PS interactions on a given thread, e.g., to determine potential causalities like loading a binary after reading its name from the Registry. In both cases, the implication is that some queries need not read timestamps at all.

4.2 PS Workloads and Log Optimizations

For our survey, we monitored the PS interactions of over 324 machines during one year across a variety of computing environments and collected over 5000 machine-days of PS interactions in total. We worked with MSN to instrument 207 of their machines, across 4 different services with different workloads, including CPU-bound systems with heavy disk workloads, a large storage service for external users, and web notifications publish/subscribe service. In our own research lab, we monitored 72 laboratory machines used for various data collection, analysis and simulation experiments. We also monitored 35 corporate desktops and laptops, used by researchers and engineers, primarily for activities such as software development and word processing. Finally, we monitored 7 home machines, used for entertainment and work-related activities by researchers, engineers, and their families. As a control, we also collected traces from 3 idle systems, running within virtual machines with no user workload.

4.2.1 Scale and repeated interactions

The primary challenge to efficiently tracing the PS interactions of a machine is the volume of events that occur. In our survey, we found that the average number of daily PS interactions was O(10⁷) ranging from 9M on desktop machines to 70M on the busiest workloads,

Table 2: The average per machine daily total and distinct interactions, entries, and processes

distinct interactions, entries, and processes									
Environment	Total PS Interactions	Distinct PS Interactions	Distinct PS Entries Accessed	Distinct Processes					
Svc. 1	70M	0.2%	>0.1%	40-60					
Svc. 4	29M	3.3%	0.4%	30-70					
Svc. 2	22M	0.6%	0.1%	30					
Svc. 3	19M	1.1%	0.1%	30-70					
Home	17M	4.2%	0.6%	30-40					
Desktop	9M	5.4%	1.0%	20-100					
Lab	9M	1.5%	0.3%	17-40					
Average	25M	1.6%	0.2%	43					
Idle	965K	2.1%	0.5%	14					

as shown in Table 2. Not surprisingly, servers tended to have a stable workload from day-to-day, while our lab, corporate desktop and home machines had varied workloads. The highest number of daily interactions we saw was 264M events, on an MSN server that collected application logs from 1000s of other machines.

However, we found several factors that mitigate the large volume of PS interactions in all these workloads. First, the number of distinct files and registry entries read or written every day is much smaller than the total number of interactions. Secondly, the number of distinct processes that run on each machine is very small, $O(10^2)$ processes on the busiest desktops, and fewer on production servers. Overall, we found that most PS entries are only accessed by a small number of processes, and that the total number of distinct interactions (*i.e.*, distinct <user, process, operation-type, PS entry> tuples) was $O(10^5)$, only 0.2% to 5.4% of the total interactions per day.

This implies that we can improve the performance of queries not interested in timestamps or counts of PS interaction occurrences by separating the unique definitions of observed interactions from the time-ordered traces of when they occur. Effectively, this allows many common queries to ignore 94.6-99.8% of the log. This also provides the possibility of compressing our logs, by replacing repeated descriptions of an interaction with a single unique ID.

4.2.2 Bursts of Activity

Several studies of I/O traffic and file system activities have shown that server and desktop I/O workloads demonstrate bursty or self-similar behavior [14,16]. We observe this in our traces as well, where it manifests as many interactions arriving together with long idle periods in between.

The primary implication of these bursts for our log format is that, when many events occur together, there is a clear opportunity to merge their associated time

information, storing a single timestamp for all the events that occur during the same timestamp bucket. This is a significant observation because per-event timestamps are a major limiting factor to achieving high compression rates. To help us choose an appropriate bucket duration, we look to the requirements of common-case systems management queries. We find that fine-grained timestamps are rarely necessary, instead what is most important is the relative ordering of events and the ability to map event occurrences to human activities (i.e., wall-clock time). This leads us to choose a relatively coarse-grained 48-bit or 6ms granularity timestamp. Note that this still provides a granularity finer than Windows' time-scheduling quantum of 10-15ms. While one might worry that a coarse-grained timestamp would mean every bucket would have at least one event in it, in practice, even our busiest observed machine-day, with 264M daily events, showed no PS interactions during 60% of its timestamp buckets. Of course, this does not mean the machine as a whole was idle—it might have been busy with CPU calculations during the times it was not doing PS interactions.

4.2.3 Repeated Sequences of Interactions

Our final key observation is that many sequences of PS interactions repeat over time. This is not a surprise, as we would expect that most of the file system and registry activities performed by a system are standard, repetitive tasks, including process start-up and shutdown, background activities, document auto-saves, and logging. We perform a detailed analysis of repeating "activity bursts," in [32] and, for space considerations, provide only a summary here.

In our analysis in [32], we define an "activity burst" as the set of PS interactions occurring in one thread, where each interaction occurs no more than some small time separation apart. Formally, we define an activity burst as a group of events $\{e_t | i \ t \ j\}$ occurring within a single thread, where $gap(e_t,e_{t+1}) < k$, for all i t < j; $gap(e_{i-1},e_i) k$; $gap(e_i,e_{i+1}) k$; gap(x,y) is the time between two interactions x and y; and k is the threshold gap between bursts. We call an activity burst a "repeat" if it is identical to an earlier activity burst in every attribute of its interactions except for timestamps. Otherwise, we call it a "distinct" burst. In our survey, we find that most activity bursts in a day are repeated bursts. On desktops, we see 2K-5K distinct bursts out of 20K-40K total and, on servers, we see 3K-4K distinct bursts out of 40K-70K.

This repetition of PS interaction sequences indicates that simple byte compression schemes, applied to the time-ordered event sequences, should detect and compress these repeating patterns. Since our analysis of activity burst repetition focuses on bursts within a single-thread, storing PS interactions in a system-wide timestamp sequence runs the risk of allowing

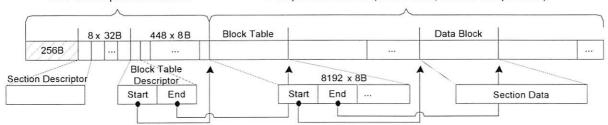


Figure 2: The physical layout of our log format

concurrent I/O from multiple threads to interfere with the compressibility of each other's patterns. However, because of the relatively large CPU time slice of 10-15ms on the Windows OS, and the knowledge that most PS interactions are handled quickly by file caches, we

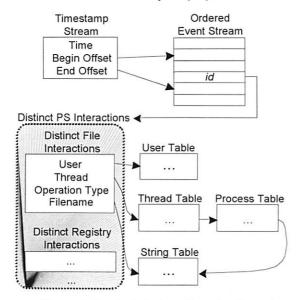


Figure 1: Logical design of our log format

still expect to gain significant compression of repeating patterns in a cross-thread trace of PS interactions.

4.3 Log Format Details

Based on the machines observed in our survey, our log format has two key facets to its logical design, shown in Figure 1. First, given the repetition of accessed files, observed processes, etc., we normalize the attributes of our persistent state interactions into separate sections, essentially following a standard procedure of database normalization.³ We create one section for distinct PS interactions, which point to other sections containing distinct names, user context, process information, file data hashes and values. The primary benefit of this normalization is a reduction of repetitive information. In addition, we find that grouping attributes into their own sections improves the

performance of byte compression algorithms as we compress these log sections later. This separation also improves query efficiency by transforming queries that involve multiple attributes and expensive string comparisons into inexpensive comparisons of integer IDs as we scan through traces of PS interactions.

Our second design choice in our logical log format is to represent the trace of events itself as two parallel, but connected, streams of data. The first stream is an ordered list of events as they are captured by our kernel driver and reported to the user daemon. The second stream contains timestamp information for groups of events. This amortizes the size of timestamp information across multiple events, reducing the overall size of the log, as well as improving byte compression of the event stream by better exposing patterns in the sequences of events. Both logical streams are stored in a single physical file to ease management of log archives.

We created a file structure that contains a large logical address space split into 32 sections. Each of the normalized attribute sections, the section of distinct PS interactions, as well as the ordered event stream and timestamp stream, are mapped to a section in our log file. Each section is composed of individually compressed 64k pages. Compressing in blocks, rather than using a streaming compression format allows random access within a data section.

To simultaneously optimize our log for random access and compression, our physical log file layout consists of a three-layer addressing scheme of block table, block number, and block offset, shown in Figure 2. This three-layer addressing scheme is important because we want to compress individual blocks and store the start and end offsets of each block in a table for fast lookup, and as a further optimization, compress these tables as well. With 448 block tables, 8192 blocks per table and a 64k uncompressed block size, this provides a maximum addressable storage size of 234 GB of uncompressed data within each log file. While we find this is many times more than a single machine-day of logs, this format gives us the flexibility of joining many days of logs into a single file for

³ Database normalization is a process of organizing data to eliminate redundancy and reduce potential for error (33).

improved compression, and gives us flexibility if PS interaction workloads grow in the future.

Each log file starts with a 4k uncompressed header. The first 256 bytes consist of versioning and other miscellaneous information. Next are 32 section descriptions, each 8 bytes long. Each of the logical sections, described earlier, is laid out contiguously over our three-layer addressing scheme, aligned at block boundaries. These section descriptors provide the block table and number of the start and end of each section. The rest of the 4k header is filled by 448 block table descriptors, that point to the start and end offsets of a compressed block table. The block table, in turn, contains 8192 block entries, each pointing to the start and end offset of a compressed 64k block.

The timestamp section is maintained as a 16 byte entry containing 6 bytes (48 bits) to represent a 6ms time resolution, 2 bytes to count missing events within that region, and two 4 byte offsets pointing to the first and last consecutive event with that time resolution. While almost all events are received by the daemon in time sorted order, we correctly handle timestamp information for any events that appear out of order. This can happen when a context switch occurs just after an I/O activity completes, but before the kernel driver reports it, but this delays the reporting of an event by a few scheduling quantums, and never affect the intra-thread ordering of PS interactions.

The user daemon first creates log files in-memory. As it receives raw events from the kernel driver, the daemon normalizes the events, replacing attribute values with indexes into the appropriate sections. The normalized event is then compared to the table of distinct normalized events using an O(1) hash lookup, and added to the table if necessary. Finally, the normalized event is added to the ordered event stream section, along with new timestamp information, if necessary. Each of the log file's data sections is append-only in memory. When a log file is closed and flushed to disk the daemon writes each data section contiguously to disk while applying a standard compression algorithm to each 64K-byte block.

4.4 Querying Logs

When analyzing these log files, our tools tend to restrict their queries based on one or more attributes in the PS interaction record, based on a time-range of interest, or based on both. To restrict a query by attribute, a query tool scans the appropriate section, looking for all values matching the given criteria. From these values, the tool then generates a filter to apply against the section of distinct PS interactions, resulting in a set of unique IDs, one for each PS interaction matching the original attribute restriction. For example, to return only PS interactions that access a particular file, a tool would first scan the string section to find the ID of the filename, and then scan the section of distinct

PS interactions to find the IDs of all distinct PS interactions that accessed this filename ID. If a tool is not interested in when or how many times an interaction occurred then it can stop here, without scanning the much larger event stream sections. Otherwise, the tool can scan through the ordered event list and timestamp stream to find the details of the occurrences of these PS interactions.

To restrict a query by a time range, a query tool applies a binary search to the timestamp stream, searching for the start of the desired time range. Once this timestamp is found, it can skip to the appropriate 64K block of the ordered list of events, and begin scanning the ordered list from that point on, until the end of the time range.

Common-case queries tend to extract sparse information from the extremely large data set of PS interactions. Our log format enables efficient queries by allowing query tools to focus on the relevant subsets of data, and expanding their scope to larger and larger portions of the data as necessary. For example, a query to find a list of all files modified during a day of 25M PS interactions requires only one pass over a distinct event table with 318Kentries to identify the string attribute id of modified files, and then scanning over the string attribute section with 100K entries to discover the full filenames of each modified file, avoiding ever scanning the full list of 25M events that occurred during the day.

5. Experimental Evaluation

In this section, we evaluate FDR and find that on average, our logs use only 0.7 bytes/PS interaction. We find that query performance scales linearly with the number of events in a log. All of our queries can be processed in a single pass against an entire machine-day of logs in just 3.2 seconds. FDR's client-side performance overhead is less than 1%, and calculating the load on bottleneck resources in our central server indicates that a single machine could scale to collecting and analyzing all the PS interactions from 4300 machines, keeping all logs for over 3 months.

5.1 Log File Compression

The efficiency of FDR's log file format affects both the runtime load on clients' memory and network, and the long-term storage costs of PS interaction logs.

In our survey of 5000 machine-days of logs, described in Section 4.2, the average raw event size is 140 bytes and daily per machine raw logs are 7GB, compressing to 700MB with GZIP. After converting our collected logs to our new format, we found, depending on workload, each PS interaction takes between 0.5 to 0.9 bytes of storage, and 0.7 bytes on average. One machine-day of PS interactions can be stored in 6 to 71MB, 20MB on average, with a

Table 3: Daily storage requirements for machines across environments

Role	Avg.	Avg.	Max	
	Bytes/Event	MB/day	MB/day	
Svc. 1	0.91	71	179	
Svc. 4	0.78	57	103	
Svc. 2	0.71	19	22	
Svc. 3	0.66	10	53	
Home	0.58	17	51	
Desktop	0.80	13	21	
Lab	0.47	6	43	
Average	0.70	20	49	
Idle	0.85	0.76	0.88	

maximum observed machine-day size of 179MB. Table 3 shows the results across our environments.

In addition to PS interaction workload, the efficiency of our log format's compression is sensitive to the frequency at which the logs are uploaded to the central server. More frequent uploads of logs reduces the latency between when an event occurs and when it can be analyzed. However, uploading logs less frequently allows the logs to collect more repeated events and achieve better overall compression.

Figure 3 shows how storage size varies with the event collection period. It shows that 80% of the compression efficiency is typically achieved with log files an hour long, and that improvement ceases after 1 week of data. Based on this result, our current deployments upload log files every 2 hours, and our central server reprocesses and merges them into 1-week long log files to save long-term storage space. We envision that future versions of FDR will support latency-critical queries by moving them to the client-side agent.

In Table 4, we look inside the log files to see how much space is taken up by each section of our log file format across our different environments. We see that the ordered event stream and timestamps dominate,

Bytes per Event vs. Log File Interval

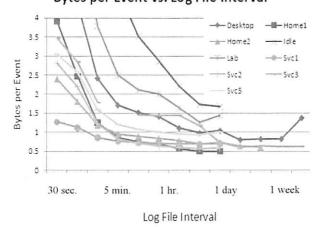


Figure 3: Compression Ratio variation with log file interval for each category of machines

Table 4: Average section size as a percentage of total log file size for each machine role

Role	Event	Time	Reg.	File	String	Data	Other
Svc 1	77%	19%	1%	0.3%	0.2%	1.9%	0.6%
Svc 4	57%	17%	7%	13.9%	1.8%	3.2%	0.1%
Svc 2	71%	15%	4%	1.4%	0.6%	7.9%	0.1%
Svc 3	69%	15%	7%	1.8%	0.9%	2.7%	3.6%
Home	46%	27%	9%	11%	4.0%	1.9%	1.1%
Desktop	47%	19%	14%	8.2%	5.2%	5.7%	0.9%
Lab	48%	31%	8%	3.5%	1.9%	7.0%	0.6%
Average	53%	22%	9%	7.7%	2.8%	4.1%	1.4%
Idle	44%	35%	7%	4.2%	4.8%	3.9%	1.5%

together taking 66-96% of the daily logs of non-idle machines. The definitions of distinct file and registry interactions are the next major contributor to log file size, taking 1.3-22.2% of daily logs from non-idle machines. Surprisingly, storing the values of every registry setting read or written (in the data section) takes only 1.9-7% of daily logs of non-idle machines.

5.2 Log File Query Performance

To evaluate FDR's query performance, we performed three types of queries against our collected daily machine logs: a single-pass query, a two-pass query, and a more complicated multi-pass query. Our first query is a single-pass scan of distinct PS interactions, and generates a manifest that identifies the distinct set of registry settings and files used by every process during the day. This query does not require scanning the time-ordered event streams. Our second query is a two-pass scan that searches for stale binaries (discussed in Section 6.1.2). This query scans through all PS interactions for files loaded for execution, and then scans the time-ordered event stream to see if the file has been modified on-disk since it was last loaded into memory, indicating that a stale copy exists in memory. Our third query looks for Extensibility Points (discussed in Section 6.2). This is our most complicated query, making multiple scans of the distinct PS interactions and time-ordered event stream.

Query performance was measured by evaluating our three queries against log files that ranged from 200k to 200M PS interactions across all categories of machines. For this experiment we used a single 3.2GHz processor. We found that the average machine-day can be queried in only 3.2 to 19.2 seconds, depending on query complexity. Figure 4 plots the count of items in scanned sections vs. the time to complete each query, and indicates that performance scales linearly with log size.

We found that our query performance was not I/O bound reading log files from disk, but rather CPU-bound, on decompressing log files. In fact, we found

Query Time vs. Scanned Log File Section Attributes

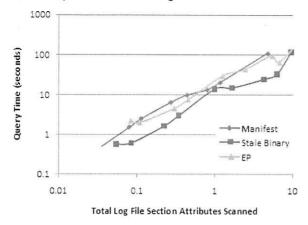


Figure 4: Time taken for each query compared with the number of attributes scanned

that query performance can be accurately predicted as a linear function of the number of items per section in the log file scanned by the query and the Intel Pentium CPU cycle cost to process each item. We measured the average per item CPU cycle cost to be 63k cycles for manifest queries, 27k for stale binary queries and 53k for extensibility point queries. Using Pearson's product moment correlation to compare our predicted and measured query times, we find the correlation to be from 0.923 to 0.998, indicating that query performance is a linear function of the size of the log items scanned.

5.3 Client-Side Overhead

Here, we evaluate the tracing and compression overhead of our kernel driver and user-mode agent. Through our initial survey of PS interactions, we have tested and deployed our driver and our user-mode agent without our compression tools on over 324 Windows 2000, Windows XP and Windows 2003 machines. At MSN, pre-production testing of our data collector was done in a lab setup of 4 identical servers, 1 running our agent, each receiving a copy of the current live production load. Measurements were made of volume and latency of workload transactions along with memory, network, CPU, and I/O overhead. The performance impact of our driver and agent was minimal, with < 1% CPU overhead measured, and no measurable degradation in transaction rate or latency. To further confirm this, we conducted an experiment where we placed a high and varied PS interaction workload, consisting of simultaneously creating 100,000 processes, scanning the entire registry and file system for non-existent entries, and making 10,000 copies of a 1KB file, 35 copies of a 1GB file, and 100,000 registry keys and values. Even under this load, we found no measurable performance degradation.

The next stage of our evaluation of agent performance focuses on the overhead of log format generation and compression. Because we configure our agent to run on a low-priority thread, we have not observed a noticeable performance impact, but do want to understand its limits. Therefore, we measure the CPU cost of processing a single PS interaction, and use this to evaluate the cost of processing our observed PS interaction rates.

Our measured CPU cost to process a PS interaction is, on average, 64k CPU cycles. This cost is constant with regard to the number of events already processed. Our highest observed spike from our collected data is 2400 interactions per second. The average peak burst every day is 1800 interactions per second. From this, we extrapolate that the highest 1 second CPU usage spike on a 3.2 GHz CPU is 64k x 2400 / 3.2 GHz = 4.8% CPU overhead. Our average peak is 3.6% CPU overhead. As a further test case, we created an artificially high PS interaction load by repeatedly accessing cached PS data, without pause, generating a rate of 15k interactions / second. Compressing these interactions at this rate produces 30% CPU overhead. Our average rate of 100-800 PS interactions per second requires only 0.2 - 1.6% CPU overhead.

5.4 Server Scalability

The scalability of a single-machine FDR server collecting and analyzing PS interaction logs from many other machines is potentially limited by several factors: network bandwidth for receiving logs, disk I/O bandwidth for storing and reading logs, CPU cost for analyzing logs, and the disk capacity for storing logs.

The single-machine configuration we consider is the one we use to collect logs from our agents today. It is a dual 3.2GHz CPU, 2GB of RAM, a 1Gbps network connection, and a 24 hard drives (400GB 7200RPM SATA) in two 12 disk RAID 5 sets with 1 parity drive, providing 8TB of storage. Assuming an average 20MB log file per machine-day, and using the performance of this system, we consider each of the potential scalability bottlenecks:

Network bandwidth: A 1Gbps network link, with an achieved bandwidth of 100 Mbps, could support 54,000 machines uploading per day.

Disk I/O bandwidth: Our RAID storage system provides 80 Mbps random access bandwidth. At this rate, we could support both writing logs and a single-pass query at a rate of 43,200 machines per day.

CPU for analysis: Following the analysis of query cost in Section 5.2, our dual processor 3.2GHz machine can support querying up to 54,000 machines per day, at a rate of 1.6 seconds per machine-day (3.2s per CPU).

From this analysis, not counting long-term storage requirements, the limiting factor to scalability appears to be disk bandwidth, supporting the centralized collection of data from 43,200 machines. Of course, this is not a sophisticated analysis, and there are likely to be issues and interactions which might further limit the scalability of a single-machine log collector. For

this reason, we apply a safety factor of 10 to our analysis, and claim that FDR can allow a single server to provide centralized collection and analysis of the complete PS interaction logs of up to 4,300 machines.

Separately, analyzing the storage capacity of our single-machine server, we find that our 8TB RAID system can store a total of 400k 20MB machine-days. This would store the complete PS interaction logs of 13,000 machine for 1 month, 4,000 machines for over 3 months, or 1000 machines for 1 year.

6. Using FDR for Systems Management

In this section, we first review our use of early ondemand tracing prototypes to attack various systems management problems, then present one new case study in detail. Throughout, we describe how each technique is improved by one or more of FDR's benefits:

Completeness: FDR gathers a complete record of reads, writes, creations, etc. to the file system and registry, including details of the running process, user account and, when appropriate, data hashes, values, etc.

Always-on: FDR's always-on tracing means that users do not have to anticipate when they might need tracing information, do not need to reproduce problems to obtain traces, and enables analysis of long-term trends.

Collection and Query Scalability: FDR's scalability eases cross-machine analysis, such as PeerPressure [35], and allows administrators to centrally apply current PS analysis techniques rigorously and en masse to large computing and IT systems.

6.1 Management Scenarios

6.1.1 Troubleshooting Misconfigurations

When a problem, like a misconfiguration, happens, troubleshooting is the task of determining what has gone wrong and fixing it. The Strider Troubleshooter [36] used a precursor to FDR, called AppTracer, to capture on-demand traces of a program's registry interactions. Once a user notices a program error, they turn on AppTracer and reproduce the problem. Strider then asks the user when the program last worked, and uses this date to find a "known-good" Windows System Restore snapshot. Strider then searches for registry settings used by the program that have changed since the known-good snapshot. With some noise filtering and ranking heuristics, Strider produces a short list of settings likely responsible for the problem.

PeerPressure [35] improves on Strider by using knowledge of the configuration settings on other machines, stored in a central database. By assuming that most other machines are configured correctly, PeerPressure removes the need for users to identify the last known-good state.

Both Strider and PeerPressure suffer from similar limitations due to their use of an on-demand AppTracer. First, both tools require users to reproduce a problem so that the AppTracer can collect a

trace—hard to accomplish if an error appears only transiently or is otherwise hard to reproduce. An always-on tracer will already have captured the trace at the time of the original problem. Secondly, both tools require the user to guess which process is failing and should be traced and the user must know to iteratively expand the scope of the on-demand tracer to debug a cross-application problem, where one process fails because of an error in another (e.g., a word processor acting as an editor for an e-mail reader). An always-on tracer will already have captured traces of all the processes on a system, obviating the user's need to guess what part of the system to trace. Finally, [35] states that updating PeerPressure's central database of machine configurations as software and operating systems are upgraded is an open challenge. With the scalable log collection provided by FDR, collecting descriptions of new configurations and software to insert into PeerPressure's central database is trivial.

Furthermore, FDR's always-on tracer improves on Strider and PeerPressure's troubleshooting in a fundamental way: whereas these previous tools are only able to locate the misconfiguration, FDR's history of PS interactions can also help place responsibility for a misconfiguration by determining when and how the misconfiguration occurred. This can help identify the root cause of the issue and prevent future occurrences.

6.1.2 Detecting Known Problems

In addition to reactive trobuleshooting, we can use always-on tracing to proactively search for specific, known problems, such as common misconfigurations and old versions of software with known vulnerabilities. One common problem, the "stale binary problem," occurs when software upgrades fail to restart affected processes or reboot a machine after replacing its on-disk binaries. The result is that the system is continuing to execute the old program inmemory. This is an especially serious problem when patching security vulnerabilities. With complete, always-on tracing, we can periodically query for the last load-time of running programs and DLLs, and compare them to their last modification-time on disk, reporting inconsistencies to system administrators.

6.1.3 Change Management: Impact Analysis

Keeping systems up-to-date with the latest security and bug patches is critical for minimizing vulnerability to malicious adversaries, and loss of productivity to software bugs. At the same time, patches can destabilize existing applications. Today, unfortunately, even if a patch only updates a single shared library, the administrators do not know in advance which applications might be affected. Consequently, patch deployment is often delayed as it undergoes a lengthy (and expensive) testing cycle, and computer systems remain vulnerable to "fixed" bugs and security holes.

To help administrators focus their testing of software upgrades, we built the Update Impact Analyzer (UIA) [10] that cross-references the files and configuration settings being patched against always-on traces of PS interactions. The UIA generates a list of programs that interact with any state that is going to be updated. Any application not on this list can be placed at a lower-priority on the testing regimen. (An exception is when a given application interacts with an updated application via inter-process communication—in this case, both applications could still require thorough testing. See Section 7.3 for a discussion of other possible candidates for always-on logging, including inter-process communication and locks.)

A primary challenge faced by UIA, as reported in [10], is that patch testing and deployment is managed centrally by administrators, but application usage, for determining the dependencies between an application and various files and settings, is distributed across many computers. FDR's scalable tracing and collection of PS interactions enables administrators to easily gather the accurate information they need.

6.1.4 Malware Mitigation

The challenges to mitigating a malware infection, whether spyware, Trojan software, viruses or worms, are detecting its existence on a system and determining how the malware entered the system. With always-on PS interaction traces, identifying running malware is a matter of querying for the hashes of loaded executables and shared libraries. Any well-known malware signatures can be flagged, and unrecognized hashes can be reported to an administrator to determine whether or not they are malicious. Following the methodology of [17], always-on tracing of PS interactions can also be analyzed to discover how malware entered a system.

To further backtrack the "route of entry" of malware. the HoneyMonkey (HM) project analyzes the PS interaction traces collected with FDR of web browsers as they visit many websites [37], Using a farm of virtual machines running scripted web browsers, HM crawls the Internet. If HM notices a web browser writing to the file system outside of its browser sandbox (e.g., writes other than temporary cache files), then it can be assured that a malicious website is exploiting a browser vulnerability to install malware. SpyCrawler, a concurrent research project, used a similar system to detect malicious websites [23]. Without FDR's detailed trace information stating which processes where making the changes, their browser monitoring system had a high false-positive rate for detecting exploits, reduced by using antivirus tools to check for known malware. New malware would not be detected.

FDR's log collection can be modified in several ways to harden it against malicious adversaries. Many malicious software programs, such as spyware bundled with otherwise legitimate downloaded software, must first be written to disk before executing. We can prevent these programs from tampering with FDR's logs after-the-fact by adding tamper-evident hash-chaining signatures [18] to our logs or by moving our user agent to a hypervisor or VM outside the accessibility of the monitored system. Malicious software that enters a system directly (e.g., via a remote buffer overflow exploit) could corrupt our kernel driver before writing to disk. To avoid this attack, the file system itself would have to be moved to a separate VM or hypervisor. Detecting malware that never interacts with the disk is outside of FDR's scope.

6.2 Case Study: Exploiting SW Extensibility

Once spyware or other malware first infects a system, they often load themselves as a plug-in or extension to the operating system, daemon, or frequently used applications such as a web browser, ensuring their continued execution on the host system. One method to defend against such malware is to monitor the settings or *extensibility points* (EPs) which control software extensions, alerting the user to changes that might signify a malware installation.

By comparing snapshots of the Windows Registry before and after 120 different malware installations, GateKeeper [38] found 34 EPs that should be monitored for signs of malicious activity. Here, we show how we can take advantage of always-on tracing to detect *potential* malware-exploitable settings, even if they are currently used only by benign software extensions. Further, we show how PS interaction traces can help rank the importance of these EPs, based on the observed privileges and lifetimes of the processes that use these extensions.

6.2.1 Detecting Extensibility Points

To discover EPs we monitor application PS interactions for files being loaded for execution⁴, and check for a previous PS interaction which contained the executable's filename. If we find such a setting, we assume that it directly triggered the executable file load and mark the setting as a *direct extensibility point*. In some cases, if we continue to search backward through the history of PS interactions, we will also find *indirect extensibility points*, where another configuration setting triggers the process to read the direct EP. For example, an indirect EP may reference an ActiveX class identifier that points to a COM⁵ object's settings that contain the executable file name.

Many EPs have a similar name prefix, indicating that plug-ins using it follow a standard design pattern. We

Our PS interaction tracing records the loading of a file for execution as a distinct activity from simply reading a file into memory.

⁵ 'Component Object Model' is a Microsoft standard for reusing and sharing software components across applications.

define a common EP name prefix as an *extensibility point class*, and the fully named EP as an *extensibility point instance*. We identify new EP classes by manually examining newly discovered EP instances that do not match an existing EP class.

To survey how significant a vulnerability EPs are, we processed 912 machine-days of traces from 53 home, desktop, and server machines. From these machines, we discovered 364 EP classes and 7227 EP instances. 6526 EP instances were direct and 701 were indirect. While 130 EP classes had only 1 instance, 28 had more than 20 unique instances. The dominant EP class consists of COM objects, and accounts for 40% of all EP instances. The next two largest EP classes are associated with the Windows desktop environment, and web browser plugins. Other popular EP classes are related to an Office productivity suite and a development environment, both of which support rich extensibility features. Overall, we found that 67% of the software programs observed in our traces accessed an EP instance, and those that did used 7 on average. Explorer.exe, responsible for the Windows desktop, used the largest number of EP classes (133 EP Classes), followed by a web browser (105 EP Classes) and an email client (73 EP Classes).

Comparing our list of EP classes with the 34 discovered by Gatekeeper, we found that our procedure detected all except 6 EPs used by programs not observed in our traces.

6.2.2 Criticality of Extensibility Points

The criticality of an EP can be estimated using 1) the privilege-level of the loading process, where higher-privilege processes such as operating system or administrator-level processes, are more critical; and 2) the lifetime of the loading process, where longer running applications provide higher availability for a malicious extension. We observed that on average a machine will have at least a third of EP instances (spanning 210 EP classes) loaded by processes that are running for 95% of the machine's uptime. We also observed that one third of all EP instances were used by processes with elevated privileges. This indicates that many EP instances are critical security hazards.

6.2.3 Lessons and Suggestions

This case study shows how FDR's traces of PS interactions can be analyzed to connect the security-sensitive behavior of loading dynamic code modules back to the critical configuration settings which control its behavior, and furthermore rank the criticality of each setting. Since this analysis requires that an EP be in use, whether by malware or by a benign software extension, FDR's scalability and always-on tracing is critical to analyzing a wide-breadth of computer usage and detecting as many EPs as possible.

Once we have discovered these EPs, we can continue to analyze their use and suggest ways to mitigate the threat from EP exposure. In particular, we observed that 44% of EP instances were not modified during our monitoring period. This suggests that system administrators could restrict write permissions on these EPs, or that application designers could transform them into static data instead of a configurable setting. Also, 70% of all EP instances were used by only single process: an opportunity for administrators or application designers to lockdown these EPs to prevent their misuse. In all, we found that only 19% of EP instances were both modified and shared by multiple applications, and thus not easy candidates for removal or lockdown. For these remaining EPs, we suggest monitoring for suspicious activities and, for critical EPs, we suggest that administrators and developers audit their usefulness vs. their potential to be misused.

7. Discussion

In this section, we discuss the implications of our work on systems management techniques, as well as limitations and opportunities for future work.

7.1 White-box and Black-Box Knowledge

Software installers use manifests to track installed software and their required dependencies, anti-virus monitors use signatures of known malware, and configuration error checkers use rules to detect known signs of misconfigurations. All of these automated or semi-automated tools use explicit prior knowledge to focus on a narrow set of state and look either for known problematic state or checking for known good state. This approach is fragile in its reliance on the correctness and completeness of pre-determined information. This information can become stale over time, might not account for all software and cannot anticipate all failures. Keeping this information complete and up-to-date is hard because of the long-tail of third-party software, in-house applications, and the continuous development of new malware. With FDRcollected traces, a black-box approach to systems management can help by augmenting predetermined information with observed truth about system behavior.

For example, today's software installers commonly fail to account for program-specific PS created post-installation, such as log files and post-install plug-ins, as well as interruptions or failures during installation or removal. These mistakes accumulate and lead to system problems.⁶ Common advice is to occasionally reinstall your computer to clean up such corruptions. Black-box tracing helps avoid this by providing installers with complete, ground-truth about installed files.

To test this approach, we compared a black-box accounting of files created during installations of 3 popular applications to the files accounted for in their

⁶ Examples of failed installations causing such problems can be found at http://support.microsoft.com/ via the article IDs: 898582, 816598, 239291, and 810932.

explicit manifests. By collecting FDR traces while installing the application, using it, and then uninstalling it, we measured the file and registry entries leaked on the system. The first application, Microsoft Office 2003, leaked no files, but did leave 1490 registry entries, and an additional 129 registry entries for each user that ran Office while it was installed. The second application, the game 'Doom3', leaked 9 files and 418 registry entries. Finally, the enterprise database Microsoft SQL Server leaked 57 files and 6 registry entries. These point examples validate our belief that predetermined information can be unreliable and that black-box analysis of FDR traces provides a more complete accounting of system behavior. Predetermined information does have its uses, however. For example, a priori knowledge can express our expectations of software behavior [27] and higher-level semantics than can be provided by a black-box analysis.

7.2 State Semantics

One of the limitations of a black-box tracing approach is that, while it can provide complete, low-level ground truth, it cannot provide any semantic guidance about the meaning of the observed activities. For example, FDR cannot tell whether the Windows Registry editor program (RegEdit) is reading a registry entry as a configuration setting to affect its own behavior, or as mere data for display. Similarly, FDR cannot tell whether any given file on disk is a temporary file, a user document, or a program binary (unless explicitly loaded for execution). Investigating how to best augment low-level tracing with heuristics and semantic, white-box knowledge is an important topic for continuing to improve systems management techniques. One option, discussed below, is to selectively log application-level events and information to expose the semantic context of lower-level PS interactions.

7.3 Logging higher-level events

The question we pose here, as a challenge for future work, is what classes of events, in addition to PS interactions, should be logged to help operators and administrators maintain reliable and secure systems?

One category, mentioned above, is the semantic context of PS interactions, such as the context we receive when software binaries must be explicitly loaded for execution. Perhaps we can receive similar context and benefit from knowing that the Windows Registry editor is reading configuration settings as data, and not to affect its own behavior. Similarly, explicitly recording whether a newly created file is a user document, temporary program state or a system file might help administrators improve backup strategies and debug problems.

A second category of higher-level events are those that help to track the provenance of data. While there has been research on how to explicitly track the providence of data, we might be able to gain some of the same benefit from simply logging a "breadcrumb" trail as new files are created. Even just integrating web browsing history with PS interactions would allow us to track the provenance of downloaded files and locate the source of malware installed via a browser exploit.

A third category of events that might benefit from FDR-style always-on logging are interactions and communications between processes, such as network connections and inter-process communication. While this category does not provide extra semantic information, these interactions are important for detecting software dependencies, fault propagation paths, and potential exposure to malware.

Altogether, extending always-on tracing to include more context and events could enable a gray-box approach to systems management, combining the benefits of black-box ground-truth and white-box semantic knowledge [2].

7.4 Using Traces to (Re) Design Programs

In this paper, we have focused on analyzing PS interactions to benefit systems administrators and operators as they attempt to understand the state of the systems they manage. However, these PS interactions might be just as useful, though on a different timescale, for developers interested in improving the applications and systems they have built. One obvious benefit is when PS interactions expose an otherwise difficult-to-track software bug. We already discussed an analysis to detect "stale binaries" after software installations (a bug in the installer). Tracing PS interactions has uncovered other bugs in several server management programs as well. Other benefits to application designers can come from specific analyses of a system's reliability and security, such as our analysis of extensibility points in Section 6.2.

The bottom-line is that always-on tracing of PS interactions improves our understanding of a system's dynamic behavior in production environments, and understanding this behavior is the first step towards improving it.

8. Conclusion

We built FDR, an efficient and scalable system for tracing and collecting a complete, always-on audit of how all running processes read, write, and perform other actions on a system's persistent state, and for scalably analyzing the enormous volume of resultant data. Thus, FDR addresses significant limitations faced by prior work in using PS interactions to solve systems management problems. We achieved our goal by designing a domain-specific log format that exploits key aspects of common-case queries of persistent state interaction workload: the relatively small number of daily distinct interactions, the burstiness of interaction occurrences, and repeated sequences of interactions.

For the last 20 years, systems management has been more of a black-art than a science or engineering discipline because we had to assume that we did not know what was really happening on our computer systems. Now, with FDR's always-on tracing, scalable data collection and analysis, we believe that systems management in the next 20 years can assume that we do know and can analyze what is happening on every machine. We believe that this is a key step to removing the "black art" from systems management.

9. Bibliography

- W. Arbaugh, W. Fithen, and J. McHugh. Windows of Vulnerability: A Case Study Analysis. In *IEEE Computer*, Vol. 33(12)
- [2] A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In SOSP, Banff, Canada, 2001
- [3] M. Baker, et al. Measurements of a Distributed File System. In SOSP, Pacific Grove, CA, 1991
- [4] N. Brownlee and K. Claffy and E. Nemeth. DNS Measurements at a Root Server. In *Global Internet Symposium*. San Antonio, TX, 2001
- [5] M. Burtscher. VPC3: A Fast and Effective Trace-Compression Algorithm. In ACM SIGMETRICS. New York, NY, 2004
- [6] Z. Chen, J. Gehrke, F. Korn. Query Optimization In Compressed Database Systems. In ACM SIGMOD. Santa Barbara, CA, 2001
- [7] E. A. Codd. A relational model for large shared databanks. *Communications of the ACM*, Vol. 13(6)
- [8] B. Cornell, P. A. Dinda, and F. E. Bustamente. Wayback: A User-level Versioning File System for Linux. In *Usenix Technical*. Boston, MA, 2004
- [9] J. Douceur, and B. Bolosky. A Large-Scale Study of File-System Content. In ACM SIGMETRICS. Atlanta, GA, 1999
- [10] J. Dunagan, et al. Towards a Self-Managing Software Patching Process Using Blacpk-box Persistent-state Manifests. In ICAC. New York, NY, 2004
- [11] G. W. Dunlap, et al. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In OSDI, Boston, MA, 2002
- [12] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In USENIX FAST, San Francisco, CA, 2003
- [13] A. Goel, et al. Forensix: A Robust, High-Performance Reconstruction System. In *ICDCS Security Workshop*. Columbus, OH, 2005
- [14] S. D. Gribble, et al. Self-similarity in File Systems. In ACM SIGMETRICS, Madison, WI, 1998
- [15] W. H. Hau, and A. J. Smith. Characteristics of I/O Traffic in Personal Compute and Server Workloads. *IBM Systems Journal.*, 347-372, Vol. 42(2)
- [16] W. W. Hsu, and A. J. Smith. Characteristics of I/O Traffic in Personal Computer and Server Workloads. UC Berkeley Computer Science Division Technical Report, UCB/CSD-02-1179, 2002
- [17] S. King, and P. Chen. Backtracking Intrusions. In SOSP, Bolton Landing, NY, 2003

- [18] L. Lamport. Password authentication with insecure communication. In *Communications of the ACM*, 24(11):770-772, Nov. 1981
- [19] J. Larus. Whole Program Paths. In PLDI, Atlanta, GA, 1999
- [20] J. Lorch, Personal communication. April 11, 2006
- [21] J. Lorch, and A. J. Smith. The VTrace Tool: Building a System Tracer for Windows NT and Windows 2000. MSDN Magazine., Vol. 15, 10
- [22] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfiguration. In ACM SIGCOMM, Pittsburgh, PA, 2002
- [23] A. Moshchuk, T. Bragin, S. D. Gribble, and H. A. Levy, Crawler-based Study of Spyware in the Web. In NDSS, San Diego, CA, 2006
- [24] D. Oppenheimer, A. Ganapathi and D. Patterson. Why do Internet services fail, and what can be done about it? In USITS. Seattle, WA, 2003
- [25] K. K. Ramakrishnan, B. Biswas, and R. Karedla. Analysis of File I/O Traces in Commercial Computing Environments. In ACM SIGMETRICS. Newport, RI, 1992
- [26] E. Rescorla. Security Holes... Who Cares? In USENIX Security Symposium. Washington, DC, 2003
- [27] P. Reynolds, et al. Pip: Detecting the unexpected in distributed systems. In NSDI, San Jose, CA, 2006
- [28] D. Roselli, J. Lorch, and T. A. Anderson. Comparison of File System Workloads. In *USENIX Technical*, San Diego, CA, 2000
- [29] C. Ruemmler, and J. Wilkes. UNIX Disk Access Patterns. In USENIX Technical, San Diego, CA, 1993
- [30] Solaris Dynamic Tracing (DTRACE) http://www.sun.com/bigadmin/content/dtrace/
- [31] C. Soules, G. Goodson, J. Strunk, G. Ganger. Metadata Efficiency in a Comprehensive Versioning File System. In USENIX FAST, San Francisco, CA, 2003
- [32] C. Verbowski, et al. Analyzing Persistent State Interactions to Improve State Management, Microsoft Research Technical Report. MSR-TR-2006-39, 2006
- [33] W. Vogels. File System Usage in Windows NT 4.0. In SOSP, Charleston, SC, 1999
- [34] H. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In ACM SIGCOMM. Portland, OR, 2004
- [35] H. Wang, et al. Automatic Misconfiguration Troubleshooting with PeerPressure. In OSDI. San Francisco, CA, 2004
- [36] Y.-M. Wang, et al. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In LISA. San Diego, CA, 2003
- [37] Y.-M. Wang, et al. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In NDSS. San Diego, CA, 2006
- [38] Y.-M. Wang, et al. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. In LISA, Atlanta, GA, 2004
- [39] A. Whitaker R. Cox, S. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In OSDI, San Francisco, CA, 2004
- [40] N. Zhu, T.-C. Chiueh. Design, Implementation, and Evaluation of Repairable File Service. In *ICDSN*, San Francisco, CA, 2003

EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors

Junfeng Yang, Can Sar, and Dawson Engler Computer Systems Laboratory Stanford University

Abstract

Storage systems such as file systems, databases, and RAID systems have a simple, basic contract: you give them data, they do not lose or corrupt it. Often they store the only copy, making its irrevocable loss almost arbitrarily bad. Unfortunately, their code is exceptionally hard to get right, since it must correctly recover from any crash at any program point, no matter how their state was smeared across volatile and persistent memory.

This paper describes EXPLODE, a system that makes it easy to systematically check real storage systems for errors. It takes user-written, potentially system-specific checkers and uses them to drive a storage system into tricky corner cases, including crash recovery errors. EXPLODE uses a novel adaptation of ideas from model checking, a comprehensive, heavy-weight formal verification technique, that makes its checking more systematic (and hopefully more effective) than a pure testing approach while being just as lightweight.

EXPLODE is effective. It found serious bugs in a broad range of real storage systems (without requiring source code): three version control systems, Berkeley DB, an NFS implementation, ten file systems, a RAID system, and the popular VMware GSX virtual machine. We found bugs in every system we checked, 36 bugs in total, typically with little effort.

1 Introduction

Storage system errors are some of the most destructive errors possible. They can destroy persistent data, with almost arbitrarily bad consequences if the system had the only copy. Unfortunately, storage code is simultaneously both difficult to reason about and difficult to test. It must always correctly recover to a valid state if the system crashes at *any* program point, no matter what data is being mutated, flushed (or not flushed) to disk, and what invariants have been violated. Further, despite the severity of storage system bugs, deployed testing methods remain primitive, typically a combination of manual inspection (with the usual downsides), fixes in reaction to bug reports (from angry users) and, at advanced sites, the alleged use of manual extraction of power cords from sockets (a harsh test indeed, but not comprehensive).

This paper presents EXPLODE, a system that makes it easy to thoroughly check real systems for such crash recovery bugs. It gives clients a clean framework to build and plug together powerful, potentially system-specific dynamic storage checkers. EXPLODE makes it easy for checkers to find bugs in crash recovery code: as they run on a live system they tell EXPLODE when to generate the disk images that could occur if the system crashed at the current execution point, which they then check for errors.

We explicitly designed EXPLODE so that clients can check complex storage stacks built from many different subsystems. For example, Figure 1 shows a version control system on top of NFS on top of the JFS file system on top of RAID. EXPLODE makes it quick to assemble checkers for such deep stacks by providing interfaces that let users write small checker components and then plug them together to build many different checkers.

Checking entire storage stacks has several benefits. First, clients can often quickly check a new layer (sometimes in minutes) by reusing consistency checks for one layer to check all the layers below it. For example, given an existing file system checker, if we can slip a RAID layer below the file system we can immediately use the file system checker to detect if the RAID causes errors. (Section 9 uses this approach to check NFS, RAID, and a virtual machine.) Second, it enables strong end-to-end checks, impossible if we could only check isolated subsystems: correctness in isolation cannot guarantee correctness in composition [22]. Finally, users can localize errors by cross-checking different implementations of a layer. If NFS works incorrectly on seven out of eight file systems, it probably has a bug, but if it only breaks on one, that single file system probably does (§9.2).

We believe EXPLODE as described so far is a worth-while engineering contribution. A second conceptual contribution is its adaptation of ideas from model checking [6, 15, 17], a typically heavyweight formal verification technique, to make its checking more systematic (and thus hopefully more effective) than a pure testing approach while remaining as lightweight as testing.

Traditional model checking takes a specification of a system (a "model") which it checks by starting from an initial state and repeatedly performing all possible actions to this state and its successors. A variety of techniques exist to make this exponential search less inefficient. Model checking has shown promise in finding

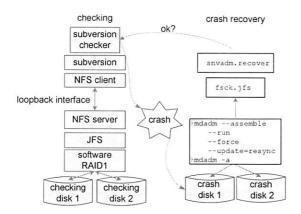


Figure 1: A snapshot of EXPLODE with a stack of storage systems being checked on the left and the recovery tools being run on the right after EXPLODE "crashes" the system to generate possible crash disks. This example checks Subversion running on top of NFS exporting a JFS file system running on RAID.

corner-case errors. However, requiring implementors to rewrite their system in an artificial modeling language makes it extremely expensive for typical storage systems (read: almost always impractical).

Recent work on *implementation-level model checking* [3, 13, 18] eliminates the need to write a model by using code itself as its own (high-fidelity) model. We used this approach in prior work to find serious errors in Linux file systems [30]. However, while more practical than a traditional approach, it required running the checked Linux system inside the model checker itself as a user-space process, which demanded enormously invasive modifications. The nature of the changes made it hard to check anything besides file systems and, even in the best case, checking a new file system took a week's work. Porting to a new Linux kernel, much less a different operating system, could take months.

This paper shows how to get essentially all the model checking benefits of our prior work with little effort by turning the checking process inside out. Instead of shoehorning the checked system inside the model checker (or worse, cutting parts of the checked system out, or worse still, creating models of the checked code) it interlaces the control needed for systematic state exploration *in situ*, throughout the checked system, reducing the modifications needed down to a single device driver, which can run inside of a lightly-instrumented, stock kernel running on real hardware. As a result, EXPLODE can thoroughly check large amounts of storage system code with little effort.

Running checks on a live, rather than emulated, system has several nice fallouts. Because storage systems already provide many management and configuration utilities, EXPLODE checkers can simply use this pre-built

machinery rather than re-implementing or emulating it. It also becomes trivial to check new storage systems: just mount and run them. Finally, any check that can be run on the base system can also be run with EXPLODE.

The final contribution of the paper is an experimental evaluation of EXPLODE that shows the following:

- EXPLODE checkers are effective (§7—§9). We found bugs in every system we checked, 36 bugs in total, typically with little effort, and often without source code (§8.1, §9.3). Checking without source code is valuable, since many robust systems rely on thirdparty software that must be vetted in the context of the integrated system.
- 2. EXPLODE checkers have enough power to do thorough checks, demonstrated by using it to comprehensively check ten Linux file systems (§7).
- 3. Even simple checkers find bugs (§8). Tiny checkers found bugs in three version control systems (§8.1) and a widely-used database (§8.2).
- 4. EXPLODE makes it easy to check subsystems designed to transparently slip into storage stacks (§9). We reused file system checkers to quickly find errors in RAID (§9.1), NFS (§9.2), and VMware (§9.3), which should not (but do) break the behavior of storage systems layered above or below them.

The paper is organized as follows. We first state our principles (§2) and then show how to use EXPLODE to check an example storage system stack (§3). We then give an overview of EXPLODE (§4) and focus on how it: (1) explores alternative actions in checked code (§5) and (2) checks crashes (§6). After the experimental evaluation (§7—§9), we discuss our experiences porting EXPLODE to FreeBSD (§ 10), contrast with related work (§11), and then conclude (§12).

2 Principles

In a sense, this entire paper boils down to the repeated application of a single principle:

Explore all choices: When a program point can legally do one of N different actions, fork execution N times and do each. For example, the kernel memory allocator can return NULL, but rarely does so in practice. For each call to this allocator we want to fork and do both actions. The next principle feeds off of this one:

Exhaust states: Do every possible action to a state before exploring another state. In our context, a state is defined as a snapshot of the system we check.

We distilled these two principles after several years of using model checking to find bugs. Model checking has a variety of tricks, some exceptionally complex. In retrospect, these capture the one feature of a model checking approach that we would take over all others: systemat-

ically do every legal action to a state, missing nothing, then pick another state, and repeat. This approach reliably finds interesting errors, even in well-tested code. We are surprised when it does not work. The key feature of this principle over traditional testing is that it makes low-probability events (such as crashes) as probable as high-probability events, thereby quickly driving the checked system into tricky corner-cases. The final two principles come in reaction to much of the pain we had with naive application of model checking to large, real systems.

Touch nothing. Almost invariably, changing the behavior of a large checked system has been a direct path to experiences that we never want to repeat. The internal interfaces of such systems are often poorly defined. Attempting to emulate or modify them produces cornercase mistakes that model checking is highly optimized to detect. Instead we try to do everything possible to run the checked system as-is and parasitically gather the information we need for checking as it runs.

Report only true errors, deterministically. The errors our system flags should be real errors, reduced to deterministic, replayable traces. All checking systems share this motherhood proclamation, but, in our context it has more teeth than usual: diagnosing even deterministic, replayable storage errors can take us over a day. The cost of a false one is enormous, as is the time needed to fight with any non-determinism.

3 How to Check a Storage System

This section shows how clients use EXPLODE interfaces to check a storage system, using a running example of a simple file system checker. Clients use EXPLODE to do two main things to a storage system. First, systematically exhaust all possibilities when the checked system can do one of several actions. Second, check that it correctly recovers from a crash. Clients can also check non-crash properties by simply inserting code to do so in either their checker or checked code itself without requiring EXPLODE support (for an example see §7.2).

Below, we explain how clients expose decision points in the checked code (§ 3.1). We then explain the three system-specific components that clients provide (written in C++). One, a *checker* that performs storage system operations and checks that they worked correctly (§3.2). Two, a *storage component* that sets up the checked system (§3.3). Finally, a *checking stack* that combines the first two into a checking harness (§3.4).

3.1 How checked code exposes choice: choose

Like prior model checkers [13,30], EXPLODE provides a function, choose, that clients use to select among possible choices in checked code. Given a program

point that has N possible actions clients insert a call "choose(N)," which will appear to fork execution N times, returning the values 0,1,...,N-1 in each child execution respectively. They then write code that uses this return value to pick one unique action out of the N possibilities. EXPLODE can exhaust all possible actions at this choose call by running all forked children. We define a code location that can pick one of several different legal actions to be a *choice point* and the act of doing so a *choice*.

An example: in low memory situations the Linux kmalloc function can return NULL when called without the __GFP_NOFAIL flag. But it rarely does so in practice, making it difficult to comprehensively check that callers correctly handle this case. We can use choose to systematically explore both success and failure cases of each kmalloc call as follows:

```
void * kmalloc(size_t size, int flags) {
  if((flags & __GFP_NOFAIL) == 0)
    if(choose(2) == 0)
    return NULL;
```

Typically clients add a small number of such calls. On Linux, we used choose to fail six kernel functions: kmalloc (as above), page_alloc (page allocator), access_ok (verify user-provided pointers), bread (read a block), read_cache_page (read a page), and end_request (indicate that a disk request completed). The inserted code mirrors that in kmalloc: a call choose(2) and an if-statement to pick whether to either (0) return an error or (1) run normally.

3.2 Driving checked code: The checker

The client provides a checker that EXPLODE uses to drive and check a given storage system. The checker implements five methods:

- mutate: performs system-specific operations and calls into EXPLODE to explore choices and to do crash checking.
- 2. check: called after each EXPLODE-simulated crash to check for storage system errors.
- 3. get_sig: an optional method which returns a byte-array signature representing the current state of the checked system. It uses domain-specific knowledge to discard irrelevant details so that EXPLODE knows when two superficially different states are equivalent and avoids repeatedly checking them. The default get_sig simply records all choices made to produce the current state.
- init and finish: optional methods to set up and clear the checker's internal state, called when EX-PLODE mounts and unmounts the checked system.

```
1 : const char *dir = "/mnt/sbd0/test-dir":
2 : const char *file = "/mnt/sbd0/test-file";
3 : static void do_fsync(const char *fn) {
4:
      int fd = open(fn, O_RDONLY);
5:
      fsync(fd);
6:
      close(fd);
7:}
8 : void FsChecker::mutate(void) {
9 :
      switch(choose(4)) {
10:
      case 0: systemf("mkdir %s%d", dir, choose(5)); break;
11:
      case 1: systemf("rmdir %s%d", dir, choose(5)); break;
      case 2: systemf("rm %s", file); break;
12:
      case 3: systemf("echo \"test\" > %s", file);
13:
14:
        if(choose(2) == 0)
15:
          sync();
16:
        else {
17:
          do_fsync(file);
18:
          // fsync parent to commit the new directory entry
19:
          do_fsync("/mnt/sbd0");
20:
21:
        check_crash_now(); // invokes check() for each crash
22:
23:
24: }
25: void FsChecker::check(void) {
26:
      ifstream in(file);
27:
28:
        error("fs", "file gone!");
29:
      char buf[1024];
30.
      in.read(buf, sizeof buf);
31:
      in.close():
32:
      if(strncmp(buf, "test", 4) != 0)
        error("fs", "wrong file contents!");
33:
34: }
```

Figure 2: Example file system checker. We omit the class initialization code and some sanity checks.

Checkers range from aggressively system-specific (or even code-version specific) to the fairly generic. Their size scales with the complexity of the invariants checked, from a few tens to many thousands of lines.

Figure 2 shows a file system checker that checks a simple correctness property: a file that has been synchronously written to disk (using either the fsync or sync system calls) should persist after a crash. Mail servers, databases and other application storage systems depend on this behavior to prevent crash-caused data obliteration. While simple, the checker illustrates common features of many checkers, including the fact that it catches some interesting bugs.

The mutate method calls choose(4) (line 9) to fork and do each of four possible actions: (1) create a directory, (2) delete it, (3) create a test file, or (4) delete it. The first two actions then call choose(5) and create or delete one of five directories (the directory name is based on choose's return value). The file creation action calls choose(2) (line 14) and forces the test file to disk using sync in one child and fsync in the other. As Figure 3 shows, one mutate call creates thirteen chil-

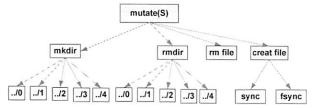


Figure 3: Choices made by one invocation of the mutate method in Figure 2's checker. It creates thirteen children.

dren.

The checker calls EXPLODE to check crashes. While other code in the system can also initiate such checking, typically it is the mutate method's responsibility: it issues operations that change the storage system, so it knows the correct system state and when this state changes. In our example, after mutate forces the file to disk it calls the EXPLODE routine check_crash_now(). EXPLODE then generates all crash disks at the exact moment of the call and invokes the check method on each after repairing and mounting it using the underlying storage component (see § 3.3). The check method checks if the test file exists (line 27) and has the right contents (line 32). While simple, this exact checker catches an interesting bug in JFS where upon crash, an fsync'd file loses all its contents triggered by the corner-case reuse of a directory inode as a file inode (§7.3 discusses a more sophisticated version of this checker).

So far we have described how a single mutate call works. The next section shows how it fits in the checking process. In addition, checking crashes at only a single code point is crude; Section 6 describes the routines EXPLODE provides for more comprehensive checking.

3.3 Setting up checked code: Storage components

Since EXPLODE checks live storage systems, these systems must be up and running. For each storage subsystem involved in checking, clients provide a storage component that implements five methods:

- 1. init: one-time initialization, such as formatting a file system partition or creating a fresh database.
- 2. mount: set up the storage system so that operations can be performed on it.
- 3. unmount: tear down the storage system; used by EXPLODE to clear the storage system's state so it can explore a different one (§5.2).
- recover: repair the storage system after an EX-PLODE-simulated crash.
- threads: return the thread IDs for the storage system's kernel threads. EXPLODE reduces nondeterminism by only running these threads when it wants to (§5.2).

```
void Ext3::init(void) {
  // create an empty ext3 FS with user-specified block size
  systemf("mkfs.ext3 -F -j -b %d %s",
     get_option(blk_size), children[0]->path());
void Ext3::recover() {
 systemf("fsck.ext3 -y %s", children[0]->path());
void Ext3::mount(void) {
  int ret = systemf("sudo mount -t ext3 %s %s",
      children[0]->path(), path());
  if(ret < 0) error("Corrupt FS: Can't mount!");</pre>
void Ext3::umount(void) {
 systemf("sudo umount %s", path());
void Ext3::threads(threads_t &thids) {
  int thid;
  if((thid=get_pid("kjournald")) != -1)
      thids.push_back(thid);
      explode_panic("can't get kjournald pid!");
```

Figure 4: Example storage component for the ext3 file system. The C++ class member children chains all storage components that a component is based on; ext3 has only one child.

Clients write a component once for a given storage system and then reuse it in different checkers. Storage systems tend to be easy to set up, otherwise they will not get used. Thus, components tend to be simple and small since they can merely wrap up already-present system commands (e.g., shell script invocations).

Figure 4 shows a storage component for the ext3 file system that illustrates these points. Its first four methods call standard ext3 commands. The one possibly non-obvious method is threads, which returns the thread ID of ext3's kernel thread (kjournald) using the expedient hack of calling the built-in EXPLODE routine get_pid which automatically extracts this ID from the output of the ps command.

3.4 Putting it all together: The checking stack

The checking stack builds a checker by glueing storage system components together and then attaching a single checker on top of them. The lowest component of a checking stack typically is a custom RAM disk (downloaded from [24] and slightly modified). While EXPLODE runs on real disks, using a RAM disk avoids non-deterministic interrupts and gives EXPLODE precise, fast control over the contents of a checked system's "persistent" storage. The simplest storage stack attaches a checker to one EXPLODE RAM disk. Such a stack does no useful crash checking, so clients typically glue one or more storage subsystems between these two. Currently a stack can only have one checker. However, there can be a fan-out of storage components, such as setting up mul-

```
// Assemble FS + RAID storage stack step by step.
void assemble(Component *&top, TestDriver *&driver) {
  // 1. load two RAM disks with size specified by user
  ekm_load_rdd(2, get_option(rdd, sectors));
  Disk *d1 = new Disk("/dev/rdd0");
  Disk *d2 = new Disk("/dev/rdd1");
  // 2. plug a mirrored RAID array onto the two RAM disks.
  Raid *raid = new Raid("/dev/md0", "raid1");
  raid->plug_child(d1);
  raid->plug_child(d2);
  // 3. plug an ext3 system onto RAID
  Ext3 *ext3 = new Ext3("/mnt/sbd0");
  ext3->plug_child(raid);
  top = ext3; // let eXplode know the top of storage stack
  // 4. attach a file system test driver onto ext3 layer
  driver = new FsChecker(ext3);
```

Figure 5: Checking stack: file system checker (Figure 2) on an ext3 file system (Figure 4) on a mirrored RAID array on two EXPLODE RAM disks. We elide the trivial class definitions Raid and Disk.

tiple RAM disks to make a RAID array. Given a stack, EXPLODE initializes the checked storage stack by calling each init bottom up, and then mount bottom up. After a crash, it calls the recover methods bottom up as well. To unmount, EXPLODE applies unmount top down. Figure 5 shows a three-layer storage stack.

4 Implementation Overview

This section gives an overview of EXPLODE. The next two sections discuss the implementation of its most important features: choice and crash checking.

The reader should keep in mind that conceptually what EXPLODE does is very simple. If we assume infinite resources and ignore some details, the following would approximate its implementation:

- 1. Create a clean initial state ($\S 3.3$) and invoke the client's mutate on it.
- 2. At every choose (N) call, fork N children.
- 3. On client request, generate all crash disks and run the client check method on them.
- 4. When mutate returns, re-invoke it.

This is it. The bulk of EXPLODE is code for approximating this loop with finite resources, mainly the machinery to save and restore the checked system so it can run one child at a time rather than an exponentially increasing number all-at-once. As a result, EXPLODE unsurprisingly looks like a primitive operating system: it has a queue of saved processes, a scheduler that picks which of these jobs to run, and time slices (that start when mutate is invoked and end when it returns). EXPLODE's scheduling algorithm: exhaust all possible combinations of choices within a single mutate call be-

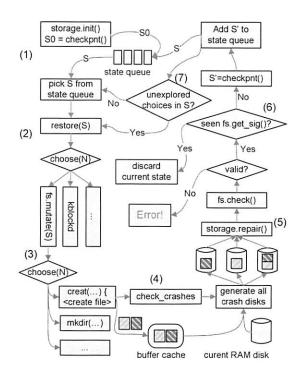


Figure 6: Simplified view of EXPLODE's state exploration loop for the file system checker in Figure 2; some choose transitions and method calls elided for space.

fore doing another (\S 2). (Note that turning EXPLODE into a random testing framework is easy: never save and restore states and make each choose(N) call return a random integer [0, N) rather than forking, recording each choice for error replay.) The above sketch glosses over some important details; we give a more accurate description below, but the reader should keep this helpful, simplistic one in mind.

From a formal method's perspective, the core of EX-PLODE is a simple, standard model checking loop based on exhausting state choices. Figure 6 shows this view of EXPLODE as applied to the file system checker of the previous section; the numbered labels in the figure correspond to the numbers in the list below:

- EXPLODE initializes the checked system using client-provided init methods. It seeds the checking process by saving this state and putting it on the state queue, which holds all states (jobs) to explore. It separately saves the created disk image for use as a pristine initial disk.
- The EXPLODE "scheduler" selects a state S from its state queue, restores it to produce a running storage system, and invokes choose to run either the mutate method or one of the checked systems' kernel threads. In the figure, mutate is selected.
- 3. mutate invokes choose to pick an action. In our example it picks creat and calls it, transferring

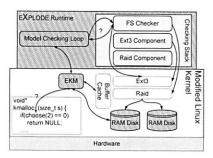


Figure 7: Snapshot: EXPLODE with Figure 5's checking stack

control to the running Linux kernel. The creat system call writes two dirty blocks to the buffer cache and returns back to mutate.

- 4. mutate calls EXPLODE to check that the file system correctly recovers from any crash at this point.
- 5. EXPLODE generates combinations of disks that could be seen after a crash. It then runs the client code to: mount the crash disk, recover it, and check it. If these methods flag an error or they crash, EXPLODE records enough information to recreate this error, and stops exploring this state.
- 6. Otherwise EXPLODE returns back into mutate which in turn returns. EXPLODE checks if it has already seen the current state using the abstracted representation returned by get_sig. If it has, it discards the state to avoid redundant work, otherwise it checkpoints it and puts it on the state queue.
- 7. EXPLODE then continues exploring any remaining choices in the original state S. If it has exhausted all choice combinations on S it picks a previously saved state off the state queue and repeats this process on it. This loop terminates when the state queue is empty or the user loses patience. (The number of possible states means the former never happens.)

After crash checking, the checked system may have a butchered internal state. Thus, before continuing, EXPLODE restores a clean copy of the current state without doing crash checking (not pictured). In addition, since checking all possible crash disks can take too long, users can set a deterministic threshold: if the number of crash disks is bigger than this threshold, EXPLODE checks a configurable number of random combinations.

Figure 7 gives a snapshot of EXPLODE; Table 1 breaks down the lines of code for each of the components. It consists of two user-level pieces: a client-provided checking stack and the EXPLODE runtime, which implements most of the model checking loop described above. EXPLODE also has three kernel-level pieces: (1) one or more RAM disks, (2) a custom kernel module, EKM, and (3) a modified Linux kernel (either version 2.6.11 or 2.6.15). EXPLODE uses EKM to monitor and determinis-

	Name	Line Count
	EKM	1,261
Linux	RAM disk Driver	326
Linux	Kernel Patch	328
	EKM-generated	2,194
	EKM	729
BSD	RAM disk Driver	357
	Kernel Patch	116
User-mode	EXPLODE	5,802
	RPC Library	521

Table 1: EXPLODE lines of code (ignoring comments and blank lines), broken down by modules. The EKM driver contains 2,194 lines of automatically generated code (EKM-generated). The EXPLODE runtime and the RPC library run at user-level, the rest is in the kernel. The RPC library is used to check virtual machines (§ 9.3). BSD counts are smaller because this port does not yet provide all EXPLODE features.

tically control checking-relevant actions done by kernel code and record system events needed for crashes. The modified kernel calls EKM to log system events and when it reaches a choice point. These modifications add 328 lines of mostly read-only instrumentation code, typically at function entry or exit. We expect them to generally be done by EXPLODE users. Unlike EXPLODE's user-space code, its RAM disk driver and EKM are kernel-specific, but are fairly small and easily ported to a new OS. We recently ported EXPLODE's core to FreeBSD, which Section 10 describes in more detail.

Given all of these pieces, checking works as follows. First, the user compiles and links their code against the EXPLODE runtime, and runs the resultant executable. Second, the EXPLODE runtime dynamically loads its kernel-level components and then initializes the storage system. Finally, EXPLODE explores the checked system's states using its model checking loop.

While checking a live kernel simplifies many things, the downside is that many bugs we find with EXPLODE cause kernel crashes. Thus, we run the checked system inside a virtual machine monitor (VMware Workstation), where it can blow itself up without hurting anyone. This approach also makes checking a non-super-user operation, with the usual benefits.

5 Exploring Choices

EXPLODE exhausts a choice point by checkpointing the current state S, exploring one choice, restoring S, and then exploring the other choices. Below we discuss how EXPLODE implements checkpoint and restore by replaying choices (§ 5.1) deterministically (§ 5.2).

5.1 Checkpointing and restoring states.

A standard checkpoint implementation would copy the current system state to a temporary buffer, which restore would then copy back. Our previous storage checking system, FiSC, did just this [30]. Unfortunately, one cannot simply save and restore a kernel running on raw hardware, so we had to instead run a heavily-hacked Linux kernel inside FiSC at user level, turning FiSC into a primitive virtual machine. Doing so was the single largest source of FiSC complexity, overhead to check new systems, and limitation on what we could check.

EXPLODE uses computation rather than copying to recreate states. It checkpoints a state S by recording the set of choices the checked code took to reach S. It restores S by starting from a clean initial state and replaying these choices. Thus, assuming deterministic actions, this method regenerates S. Mechanically, checkpoint records the sequence of n choices that produced S in an array; during replay the ith choose call simply returns the ith entry in this array.

This one change led to orders of magnitude reduction in complexity and effort in using EXPLODE as opposed to FiSC, to the degree that EXPLODE completely subsumes our prior work in almost every aspect by a large amount. It also has the secondary benefit that states have a tiny representation: a sequence of integers, one for each choice point, where the integer specifies which of N choices were made. Note that some model checkers (and systems in other contexts [10]) already use replay-recreation of states, but for error reporting and state size reduction, rather than for reducing invasiveness. One problem with the approach is that the restored state's timestamps will not match the original, making it harder to check some time properties.

Naively, it might seem that to reset the checked systems' state we have to reboot the machine, re-initialize the storage system, mount it, and only then replay choices. This expensive approach works, but fortunately, storage systems have the observed, nice property that simply unmounting them clears their in-memory state, removing their buffer cache entries, freeing up their kernel data structures, etc. Thus, EXPLODE uses a faster method: call the client-supplied unmount to clear the current state, then load a pristine initial state (saved after initialization) using the client-supplied mount.

It gets more costly to restore states as the length of their choice sequence grows. Users can configure EX-PLODE to periodically chop off the prefix of choice sequences. It does so by (1) calling unmount to force the checked system state to disk and (2) using the resultant disk image as a new initial state that duplicates the effect of the choices before the unmount call. The downside is that it can no longer reorder buffer cache entries from before this point during crash checking.

5.2 Re-executing code deterministically

EXPLODE's restore method only works if it can deterministically replay checked code. We discuss how EXPLODE does so below, including the restrictions imposed on the checked system.

Doing the same choices. Kernel code containing a choose call can be invoked by non-checking code, such as interrupt handlers or system calls run by other processes. Including such calls makes it impossible to replay traces. EXPLODE filters them by discarding any calls from an interrupt context or calls from any process whose ID is not associated with the checked system.

Controlling threads. EXPLODE uses priorities to control when storage system threads run (§ 4, bullet 2). It quiesces storage system threads by giving them the lowest priority possible using an EKM ioctl. It runs a thread by giving it a high priority (others still have the lowest) and calling the kernel scheduler, letting it schedule the right thread. It might seem more sensible for EX-PLODE to orchestrate thread schedules via semaphores. However, doing so requires intrusive changes and, in our experience [30], backfires with unexpected deadlock since semaphores prevent a given thread from running even if it absolutely must. Unfortunately, using priorities is not perfect either, and still allows non-deterministic thread interleaving. We detect pathological cases where a chosen thread does not run, or other "disabled" threads do run using the "last-run" timestamps in the Linux process data structure. These sanity checks let us catch when we generate an error trace that would not be replayable or when replaying it takes a different path. Neither happens much in practice.

Requirements on the checked system. The checked system must issue the same choose calls across replay runs. However, many environmental features can change across runs, providing many sources of potential non-deterministic input: thread stacks in different locations, memory allocations that return different blocks, data structures that have different sizes, etc. None of these perturbations should cause the checked code to behave differently. Fortunately, the systems we checked satisfy this requirement "out of the box" - in part because they are isolated during checking, and nothing besides the checker and their kernel threads call into them to modify their RAM disk(s). Non-deterministic systems require modification before EXPLODE can reliably check them. However, we expect such cases to rarely occur. If nothing else, usability forces systems to ensure that reexecuting the same user commands produces the same system state. As a side-effect, they largely run the same code paths (and thus would hit the same choose calls).

While checked code must do the same choose calls for deterministic error replay, it does not have to allocate the same physical blocks. EXPLODE replays choices, but then regenerates all different crash combinations after the last choice point until it (re)finds one that fails checking. Thus, the checked code can put logical contents in different physical blocks (e.g., an inode resides in disk block 10 on one run and in block 20 on another) as long as the logical blocks needed to cause the error are still marked as dirty in the buffer cache.

6 Checking Crashes

This section discusses crash checking issues: EX-PLODE's checking interface (\S 6.1), how it generates crash disks (\S 6.2), how it checks crashes during recovery (\S 6.3), how it checks for errors caused by application crashes (\S 6.4), and some refinements (\S 6.5).

6.1 The full crash check interface

The check_crashes_now() routine is the simplest way to check crashes. EXPLODE also provides a more powerful (but complex) interface clients can use to directly inspect the log EXPLODE extracts from EKM. They can also add custom log records. Clients use the log to determine what state the checked system should recover to. They can initiate crash checking at any time while examining the log. For space reasons we do not discuss this interface further, though many of our checkers use it. Instead we focus on two simpler routines check_crashes_end that give most of the power of the logging approach.

Clients call check_crashes_start before invoking the storage system operations they want to check and check_crashes_end after. For example, assume we want to check if we can atomically rename a file A to B by calling rename and then sync(). We could write the following code in mutate:

// Assume: A, B on disk	Legal state(s) after crash
check_crashes_start();—	(A and B)
rename("A", "B"); sync();	(A and B), or B
check_crashes_end(); —	В

EXPLODE generates all crash disks that can occur (inclusively) between these calls, invoking the client's check method on each. Note how the state the system should recover to changes. At the check_crashes_start call, the recovered file system should contain both A and B. During the process of renaming, the recovered file system can contain either (1) the original A and B or (2) B with A's original contents. After sync completes, only B with A's original contents should exist.

This pattern of having an initial state, a set of legal intermediate states, and a final state is a common one for checking. Thus, EXPLODE makes it easy for check to distinguish between these epochs by passing a flag that tells check if the crash disk could occur at the first call (EXP_BEGIN), the last call (EXP_END), or in between (EXP_INBETWEEN). We could write a check method to use these flags as follows:

```
check(int epoch, ...) {
   if(epoch == EXP_BEGIN)
      // check (A and B)
   else if(epoch == EXP_INBETWEEN)
      // check (A and B) or B
   else // EXP_END
      // check B
}
```

EXPLODE uses C++ tricks so that clients can pass an arbitrary number of arguments to these two routines (up to a user-specified limit) that in turn get passed to their check method.

6.2 Generating crash disks

EXPLODE generates crash disks by first constructing the current *write set*: the set of disk blocks that currently *could be* written to disk. Linux has over ten functions that affect whether a block can be written or not. The following two representative examples cause EXPLODE to add blocks to the write set:

- mark_buffer_dirty(b) sets the dirty flag of a block b in the buffer cache, making it eligible for asynchronous write back.
- generic_make_request(req) submits a list of sectors to the disk queue. EXPLODE adds these sectors to the write set, even if they are clean, which can happen for storage systems maintaining their own private buffer caches (as in the Linux port of XFS).

The following three representative examples cause EX-PLODE to remove blocks from the write set:

- clear_buffer_dirty(b) clears b's dirty flag.
 The buffer cache does not write clean buffers to disk.
- end_request(), called when a disk request completes. EXPLODE removes all versions of the request's sectors from the write set since they are guaranteed to be on disk.
- lock_buffer(b), locks b in memory, preventing it from being written to disk. A subsequent clear_buffer_locked(b) will add b to the write set if b is dirty.

Writing any subset of the current write set onto the current disk contents generates a disk that could be seen if the system crashed at this moment. Figure 8 shows how EXPLODE generates crash disks; its numbered labels correspond to those below:

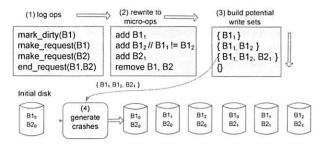


Figure 8: Generating all potential crash disks.

- As the storage system executes, EKM logs operations that affect which blocks could be written to disk.
- EXPLODE extracts this log using an EKM ioctl and reduces the logged operations to micro-operations that add or remove blocks from the write set.
- 3. It then applies these add and remove operations, in order, to the initial write set.
- Whenever the write set shrinks, it generates all possible crash disks by applying all subsets of the write set to the current disk. (Doing so when the write set shrinks rather than grows makes it trivial to avoid duplicate work.)

Note that the write set tracks a block's contents in addition to the block itself. Naively it may appear that when EXPLODE adds a block b to the write set it should replace any previous copy of b with this more recent one. (Our previous work [30] did exactly this.) However, doing so misses errors. For example, in the figure, doing so misses one crash disk $(B1_1, B2_1)$ since the second insertion of block B1 replaces the previous version $B1_1$ with $B1_2$.

6.3 Checking crashes during recovery

Clients can also use EXPLODE to check that storage systems correctly handle crashes during recovery. Since environmental failures are correlated, once one crash happens, another is not uncommon: power may flicker repeatedly in a storm or a machine may keep rebooting because of a bad memory board. EXPLODE generates the disks that could occur if recovery crashes, by tracking the write set produced while running recover, and then applying all its subsets to the initial crash disk. It checks these "crash-crash" disks as it would a crash disk. Note this assumes recovery is idempotent in that if a correct recovery with no crash produces state S_{valid} then so should a prematurely crashed repair followed by a successful one. We do not (but could) check for further crashes during recovery since implementors seem uninterested in such errors [30].

6.4 Checking "soft" application crashes

In addition to "hard" machine crashes that wipe volatile state, EXPLODE can also check that applications cor-

rectly recover from "soft" crashes where they crashed, but the operating system did not. Such soft crashes are usually more frequent than hard crashes with causes ranging from application bugs to impatient users pressing "ctrl-C." Even applications that ignore hard crashes should not corrupt user data because of a soft crash.

EXPLODE checks soft crashes in two steps. First, it runs the checker's mutate method and logs all mutating file system operations it performs. Second, for each log prefix EXPLODE mounts the initial disk and replays the operations in the prefix in the order they are issued. If the log has n operations EXPLODE generates n storage states, and passes each to the check method.

6.5 Refinements

In some cases we remove blocks from the write set too eagerly. For example, we always remove the sectors associated with end_request, but doing so can miss permutations since subsequent writes may not in fact have waited for (depended on) the write to complete. Consider the events: (1) a file system writes sector S1, (2) the write completes, (3) it then writes sector S2. If the file system wrote S2 without explicitly waiting for the S1 write to complete then these writes could have been reordered (i.e., there is no happens-before dependency between them). However, we do not want to grovel around inside storage systems rooting out these false dependencies, and conservatively treat all writes that complete as waited for. A real storage system implementor could obviously do a better job.

To prevent the kernel from removing buffers from the write set, we completely disable the dirty buffer flushing threads pdflush, and only schedule the kernel thread kblockd that periodically flushes the disk queue between calls to the client mutate method.

If a checked system uses a private buffer cache, EX-PLODE cannot see all dirty blocks. We partially counter this problem by doing an unmount before generating crash disks, which will flush all private dirty buffers to disk (when EXPLODE can add them to its write set). Unfortunately, this approach is not a complete solution since these unmount-driven flushes can introduce spurious dependencies (as we discussed above).

7 In-Depth Checking: File Systems

This section demonstrates that EXPLODE's lightweight approach does not compromise its power by replicating (and sometimes superseding) the results we obtained with our previous, more strenuous approach [30]. It also shows EXPLODE's breadth by using it to check ten Linux file systems with little incremental effort.

We applied EXPLODE to all but one of the disk based

file systems on Linux 2.6.11: ext2, ext3, JFS, ReiserFS, Reiser4, XFS, MSDOS, VFAT, HFS, and HFS+. We skipped NTFS because repairing a crashed NTFS disk requires mounting it in Windows. For most file systems, we used the most up-to-date utilities in the Debian "etch" Linux distribution. For HFS and HFS+, we had to download the source of their utilities from OpenDarwin [14] and compile it ourselves. The storage components for these file systems mirror ext3's component (§ 3.3). Four file systems use kernel threads: JFS, ReiserFS, Reiser4 and XFS. We extracted these thread IDs using the same trick as with ext3.

While these file systems vary widely in terms of implementation, they are identical in one way: none give clean, precise guarantees of the state they recover to after a crash. As a result, we wrote three checkers that focused on different special cases where what they did was somewhat well-defined. We built these checkers by extending a common core, which we describe below. We then describe the checkers and the bugs they found.

7.1 The generic checker core

The basic checker starts from an empty file system and systematically generates file system topologies up to a user-specified number of files and directories. Its mutate exhaustively applies each of the following eight system calls to each node (file, link, directory) in the current topology before exploring the next: ftruncate, pwrite (which writes to a given offset within a file), creat, mkdir, unlink, rmdir, link and rename. For example, if there are two leaf directories, the checker will delete both, create files in both, etc. Thus, the number of possible choices for a given tree grows (deterministically) with its size. For file systems that support holes, the checker writes at large offsets to exercise indirect blocks. Other operations can easily be added.

For each operation it invokes, mutate duplicates its effect on a fake "abstract" file system it maintains privately. For example, if it performs three operations mkdir(/a), mkdir(/a/b), and sync() then the abstract file system will be the tree /a/b, which the real file system must match exactly. The checker's get_sig method returns a canonical version of this abstract file system. This canonicalization mirrors that in [30], and uses relabeling to make topologies differing only in naming equivalent and discards less interesting properties such as timestamps, actual disk blocks used, etc.

7.2 Check: Failed system calls have no effect

This check does not involve crash-recovery. It checks that if a file system operation (except pwrite) returns an error, the operation has no user-visible effect. It uses

EXPLODE to systematically fail calls to the six kernel functions discussed in Section 3.1. The actual check uses the abstract file system described in the previous subsection. If a system call succeeds, the checker updates the abstract file system, but otherwise does not. It then checks that the real file system matches the abstract one.

Bugs found. We found 2 bugs in total. One of them was an unfixed Linux VFS bug we already reported in [30]. The other one was a minor bug in ReiserFS ftruncate which can fail with its job half-done if memory allocation fails. We also found that Reiser4 calls panic on memory allocation failures, and ReiserFS calls panic on disk read failures. (We did not include these two undesired behaviors in our bug counts.)

7.3 Check: "sync" operations work

Applications such as databases and mail servers use operating system-provided methods to force their data to disk in order to prevent crashes from destroying or corrupting it. Unfortunately, they are completely at these routines' mercy — there is no way to check they do what they claim, yet their bugs can be almost arbitrarily bad.

Fortunately, EXPLODE makes it easy to check these operations. We built a checker (similar to the one in Figure 2) to check four methods that force data to disk:

- 1. sync forces all dirty buffers to disk.
- 2. fsync(fd) forces fd's dirty buffers to disk.
- 3. Synchronously mounted file system: a system call's modifications are on disk when the call returns.
- Files opened with O_SYNC: all modifications done by a system call through the returned file descriptor are on disk when the call returns.

After each operation completes and its modifications have been forced to disk, the sync-checker tells EX-PLODE to do crash checking and verifies that the modifications persist.

Note, neither fsync nor O_SYNC guarantee that directory entries pointing to the sync'd file are on disk, doing so requires calling fsync on any directory containing the file (a legal operation in Linux). Thus, the checker does an fsync on each directory along the path to the sync'd file, ensuring there is a valid path to it in the recovered file system.

Bugs found. Table 2 summarizes the 13 bugs found with this checker. Three bugs show up in multiple ways (but are only counted three times): a VFS limitation caused all file systems to fail the O_SYNC check, and both HFS and HFS+ mangled file and directory permissions after crashing, therefore failing all four sync checks. We describe a few of the more interesting bugs below.

Besides HFS/HFS+, both MSDOS and VFAT mishandled sync. Simple crashes after sync can introduce di-

FS	sync	mount sync	fsync	O_SYNC
ext2		X	X	×
ext3				×
ReiserFS		×		×
Reiser4				×
JFS		×	×	×
XFS		×	74-4	×
MSDOS	×	×		×
VFAT	×	×		×
HFS	×	×	×	×
HFS+	×	×	×	×

Table 2: Sync checking results: **X** indicates the file system failed the check. There were 13 bugs, three of which show up more than once, causing more **X** marks than errors.

rectory loops. The maintainers confirmed they knew of these bugs, though they had not been publicly disclosed. These bugs have subsequently been fixed. Eight file systems had synchronous mount bugs. For example, ext2 gives no consistency guarantees by default, but mounting it synchronously still allows data loss.

There were two interesting fsync errors, one in JFS (§3.2) and one in ext2. The ext2 bug is a case where an implementation error points out a deeper design problem. The bug occurs when we: (1) shrink a file "A" with truncate and (2) subsequently creat, write, and fsync a second file "B." If file B reuses the indirect blocks of A freed via truncate, then following a crash e2fsck notices that A's indirect blocks are corrupt and clears them, destroying the contents of B. (For good measure it then notices that A and B share blocks and "repairs" B by duplicating blocks from A.) Because ext2 makes no guarantees about what is written to disk, fundamentally one cannot use fsync to safely force a file to disk, since the file can still have implicit dependencies on other file system state (in our case if it reuses an indirect blocks for a file whose inode has been cleared in memory but not on disk).

7.4 Check: a recovered FS is "reasonable"

Our final check is the most stringent: after a crash a file system recovers to a "reasonable" state. No files, directories, or links flushed to disk are corrupted or disappear (unless explicitly deleted). Nor do they spontaneously appear without being created. For example, if we crash after performing two operations mkdir(/A) and mkdir(/A/B) on an empty file system, then there are exactly three correct recovered file systems: (1) / (no data), (2) /A, or (3) /A/B. We should not see directories or files we never created. Similarly, if /A was forced to disk before the crash, it should still exist.

For space reasons we only give a cursory implementation overview. As mutate issues operations, it builds two sets: (1) the stable set, which contains the operations it knows are on the disk, (2) the volatile set, which

contains the operations that may or may not be on disk. The check method verifies that the recovered file system can be constructed using some sequence of volatile operations legally combined with all the stable ones. The implementation makes heavy use of caching to prune the search and "desugars" operations such as mkdir into smaller atomic operations (in this case it creates an inode and then forms a link to it) to ensure it can describe their intermediate effects.

Bugs found. We applied this check to ext2, ext3, JFS, ReiserFS and Reiser4. Unsurprisingly, since ext2 gives no crash guarantees, files can point to uninitialized blocks, and sync'd files and directories can be removed by its fsck. Since JFS journals metadata but not data, its files can also point to garbage. These behaviors are design decisions so we did not include them in our bug counts. We found two bugs (one in JFS, one in Reiser4) where crashed disks cannot be recovered by fsck. We could not check many topologies for ReiserFS and Reiser4 because they appear to leak large amounts of memory on every mount and unmount (Our bug counts do not include these leaks.)

In addition, we used the crash-during-recovery check (§6.3) on Reiser4. It found a bug where Reiser4 becomes so corrupted that mounting it causes a kernel panic. (Since our prior work explored this check in detail we did not apply it to more systems.)

Finally, we did a crude benchmark run by running the checker (without crash-during-recovery checking) to ext3 inside a virtual machine with 1G memory on a Intel P4 3.2GHZ with 2G memory. After about 20 hours, EXPLODE checked 230,744 crashes for 327 different FS topologies and 1582 different FS operations. The run died because Linux leaks memory on each mount and unmount and runs out of memory. Although we fixed two leaks, more remain (we did not count these obliquely-detected errors in our bug counts but were tempted to). We intend to have EXPLODE periodically checkpoint itself so we can reboot the machine and let EXPLODE resume from the checkpoints.

8 Even Simple Checkers Find Bugs

This section shows that even simple checkers find interesting bugs by applying it to three version control systems and the Berkeley DB database.

The next two sections demonstrate that EXPLODE works on many different storage systems by applying it to many different ones. The algorithm for this process: write a quick checker, use it to find a few errors, declare success, and then go after another storage system. In all cases we could check many more invariants. Table 3 summarizes all results.

System	Storage	Checker	Bugs
FS	744	5,477	18
CVS	27	68	1
Subversion	-	-	1
EXPENSIV	30	124	3
Berkeley DB	82	202	6
RAID	144	FS + 137	2
NFS	34	FS	4
VMware GSX/Linux	54	FS	1
Total	1,115	6,008	36

Table 3: Summary of all storage systems checked. All line counts ignore comments and whitespace. Storage gives the line count for each system's storage component, which for FS includes the components for all ten file systems. Checker gives the checker line counts, which for EXPENSIV includes two checkers. We reused the FS checker to check RAID, NFS and VMware. We wrote an additional checker for RAID. We checked Subversion using an early version of EXPLODE; we have not yet ported its component and checker.

8.1 Version control software

This section checks three version control systems: CVS, Subversion [27], and an expensive commercial system we did not have source code for, denoted as EXPENSIV (its license precludes naming it directly). We check that these systems meet their fundamental goal: do not lose or corrupt a committed file. We found errors in all three.

The storage component for each wraps up the commands needed to set up a new repository on top of one of the file systems we check. The checker's mutate method checks out a copy of the repository, modifies it, and commits the changes back to the main repository. After this commit completes, these changes should persist after any crash. To test this, mutate immediately calls check_crashes_now() after the commit completes. The check method flags an error if: (1) the version control systems' crash recovery tool (if any) gives an error or (2) committed files are missing.

Bugs found. All three systems made the same mistake. To update a repository file A without corrupting it, they first update a temporary file B, which they then atomically rename to A. However, they forget to force B's contents to disk before the rename, which means a crash can destroy it.

In addition EXPENSIV purports to atomically merge two repositories into one, where any interruption (such as crash) will either leave the two original repositories or one entirely (correctly) merged one. EXPLODE found a bug where a crash during merge corrupts the repository, which EXPENSIV's recovery tool (EXPENSIV -r check -f) cannot fix. This error seems to be caused by the same renaming mistake as above.

Finally, we found that even a soft crash during a merge corrupts EXPENSIV's repository. It appears EXPENSIV renames multiple files at the end of the merge. Although

each individual rename is atomic against a soft crash, their aggregation is not. The repository is corrupted if not all files are renamed.

8.2 Berkeley DB

The database checker in this section checks that after a crash no committed transaction records are corrupted or disappear, and no uncommitted ones appear. It found six bugs in Berkeley DB 4.3 [2].

Berkeley DB's storage component only defines the init method, which calls Berkeley DB utilities to create a database. It does not require mount or unmount, and has no threads. It performs recovery when the database is opened with the DB_RECOVER flag (in the check method). We stack this component on top of a file system one.

The checker's mutate method is a simple loop that starts a transaction, adds several records to it, and then commits this transaction. It records committed transactions. It calls check_crashes_start before each commit and check_crashes_end (§ 6.1) after to verify that there is a one-to-one mapping between the transactions it committed and those in the database.

Bugs found. We checked Berkeley DB on top of ext2, ext3, and JFS. On ext2 creating a database inside a transaction, while supposedly atomic, can lead to a corrupted database if the system crashes before the database is closed or sync is manually called. Furthermore, even with an existing database, committed records can disappear during a crash. On ext3 an unfortunate crash while adding a record to an existing database can again leave the database in an unrecoverable state. Finally, on all three file systems, a record that was added but never committed can appear after a crash. We initially suspected these errors came from Berkeley DB incorrectly assuming that file system blocks were written atomically. However, setting Berkeley DB to use sector-aligned writes did not fix the problem. While the errors we find differ depending on the file system and configuration settings, some are probably due to the same underlying problem.

9 Checking "Transparent" Subsystems

Many subsystems transparently slip into a storage stack. Given a checker for the original system, we can easily check the new stack: run the same checker on top of it and make sure it gives the same results.

9.1 Software RAID

We ran two checkers on RAID. The first checks that a RAID transparently extends a storage stack by running the file system sync-checker (§ 7.3) on top of it. A file system's crash and non-crash behavior on top of RAID

should be the same as without it: any (new) errors the checker flags are RAID bugs. The second checks that losing any single sector in a RAID1 or RAID5 stripe does not cause data loss [20]. I.e., the disk's contents were always correctly reconstructed from the non-failed disks.

We applied these checks to Linux's software RAID [26] levels 1 and 5. Linux RAID groups a set of disks and presents them as a single block device to the rest of the system. When a block write request is received by the software RAID block device driver, it recomputes the parity block and passes the requests to the underlying disks in the RAID array. Linux RAID repairs a disk using a very simple approach: overwrite all of the disk's contents, rather than just those sectors that need to be fixed. This approach is extremely slow, but also hard to mess up. Still, we found two bugs.

The RAID storage component methods map directly to different options for its administration utility mdadm. The init method uses mdadm --create to assemble either two or four RAM disks into a RAID1 or RAID5 array respectively. The mount method calls mdadm --assemble on these disks and the unmount method tears down the RAID array by invoking mdadm --stop. The recover method reassembles and recovers the RAID array. We used the mdadm --add command to replace failed disks after a disk failure. The checking stack is similar to that in Figure 5.

Bugs found. The checker found that Linux RAID does not reconstruct the contents of an unreadable sector (as it easily could) but instead marks the *entire* disk that contains the bad sector as faulty and removes it from the RAID array. Such a fault-handling policy is not so good: (1) it makes a trivial error enough to prevent the RAID from recovering from *any* additional failure, and (2) as disk capacity increases, the probability that another sector goes bad goes to one.

Given this fault-handling policy, it is unsurprising our checker found that after two sector read errors happen on different disks, requiring manual maintenance, almost all maintenance operations (such as mdadm --stop or mdadm --add) fail with a "Device or resource busy" error. Disk write requests also fail in this case, rendering the RAID array unusable until the machine is rebooted. One of the main developers confirmed that these behaviors were bad and should be fixed with high priority [4].

9.2 NFS

NFS synchronously forces modifications to disk before requests return [23]. Thus, with only a single client modifying an NFS file system, after a crash NFS must recover to the same file system tree as a local file system mounted synchronously. We check this property by running the

sync-checker (§7.3) on NFS and having it treat NFS as a synchronously mounted file system. This check found four bugs when run on the Linux kernel's NFS (NFSv3) implementation [19].

The NFS storage component is a trivial 15-lines of code (plus a hand-edit of "/etc/exports" to define an NFS mount point). It provides two methods: (1) mount, which sets up an NFS partition by exporting a local FS over the NFS loop-back interface and (2) unmount, which tears down an NFS partition by unmounting it. It does not provide a recover method since the recover of the underlying local file system must be sufficient to repair crashed NFS partitions. We did not model network failures, neither did we control the scheduling of NFS threads, which could make error replay non-deterministic (but did not for ours).

Bugs found. The checker found a bug where a client that writes to a file and then reads the same file through a hard link in a different directory will not see the values of the first write. We elide the detailed cause of this error for space, other than noting that diagnosing this bug as NFS's fault was easy, because it shows up regardless of the underlying file system (we tried ext2, ext3, and JFS).

We found additional bugs specific to individual file systems exported by NFS. When JFS is exported over NFS, the link and unlink operations are not committed synchronously. When an ext2 file system is exported over NFS, our checker found that many operations were not committed synchronously. If the NFS server crashes these bugs can lose data and cause data values to go backwards for remote clients.

9.3 VMware GSX server

In theory, a virtual machine slipped beneath a guest OS should not change the crash behavior of a correctly-written guest storage system. Roughly speaking, correctness devolves to not lying about when a disk block actually hits a physical disk. In practice, speed concerns make lying tempting. We check that a file system on top of a virtual machine provided "disk" has the same synchronous behavior as running without it (again) using the sync-checker (§7.3). We applied this check to VMware GSX 3.2.0 [29] running on Linux. GSX is an interesting case for EXPLODE: a large, complex commercial system (for which we lack source code) that, from the point of view of a storage system checker, implements a block device interface in a strange way.

The VMware GSX scripting API makes the storage component easy to build. The init method copies a precreated empty virtual disk image onto the file system on top of EXPLODE RAM disk. The mount method starts the virtual machine using the command

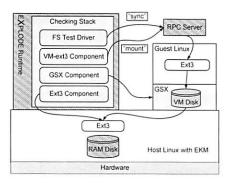


Figure 9: The VMware checking stack.

vmware-cmd start and unmount stops it using vmware-cmd stop hard. The recover method calls vmware-cmd start, which repairs a crashed virtual machine, and then removes a dangling lock (created by the "crashed" virtual machine to prevent races on the virtual disk file).

As shown in Figure 9 the checking stack was the most intricate of this paper. It has five layers, starting from bottom to top: (1) a RAM disk, (2) the ext3 file system in the host, storing the GSX virtual disk file, (3) GSX, (4) the ext3 file system in the guest, (5) the sync-checker. The main complication in building this stack was the need to split EXPLODE into two pieces, one running in the host, the other in the guest. Since the virtual machine will frequently "crash" we decided to keep the part running inside it simple and make it a stateless RPC server. The entire storage stack and the sync-checker reside in the host. When the sync-checker wants to run an operation in the guest, or a storage method wants to run a utility, they do RPC calls to the server in the guest, which then performs the operation.

Bugs found. Calling sync in the guest OS does not correctly flush dirty buffers to disk, but only to the host's buffer cache. According to VMware documents, setting the "disable write caching" configuration flag forces all writes to disk. However, we hit the same bug even with this flag on. This bug makes it impossible to reliably run a storage system on top of this VMM on Linux. We confirmed this problem with one of the main developers who stated that it should not show up in the latest version [28].

10 Checking on a new system: FreeBSD

We ported EXPLODE to FreeBSD 6.0 to ensure porting was easy and to shake out Linux-specific design assumptions. We spent most of our time writing a new RAM disk and EKM module; we only needed to change a few lines in the user-level runtime to run on FreeBSD.

The FreeBSD version of EXPLODE supports crash checking, but currently does not provide a kernel-level choose nor logging of system calls. Neither should

present a challenge here or in general. Even without these features, we reproduced the errors in CVS and EXPENSIV we saw on Linux as well as finding new errors in FreeBSD UFS2. Below, we discuss issues in writing EKM and the RAM disk.

EKM. Crash checking requires adding calls to EKM in functions that mark buffers as clean, dirty, or write them to disk. While a FreeBSD developer could presumably enumerate all such functions easily, our complete lack of experience with FreeBSD meant it took us about a week to find all corner-cases. For example, FreeBSD's UFS2 file system sometimes bypasses the buffer cache and writes directly to the underlying disk.

There were also minor system-differences we had to correct for. As an example, while Linux and FreeBSD have similar structures for buffers, they differ in how they store bookkeeping information (e.g., representing offsets in sectors on Linux, and in bytes on FreeBSD). We adjusted for such differences inside EKM so that EX-PLODE's user-level runtime sees a consistent interface. We believe porting should generally be easy since EKM only logs the offset, size, and data of buffer modifications, as well as the ID of the modifying thread. All of these should be readily available in any OS.

RAM disk. We built our FreeBSD RAM disk by modifying the /dev/md memory-based disk device. We expect developers can generally use this approach: take an existing storage device driver and add trivial ioctl commands to read and write its disk state by copying between user- and kernel-space.

Bug-Finding Results. In addition to our quick tests to replicate the EXPENSIV and CVS bugs, we also ran our sync-checker (§7.3) on FreeBSD's UFS2 with soft updates disabled. It found errors where fsck with the -p option could not recover from crashes. While fsck without -p could repair the disk, the documentation for fsck claims -p can recover from all errors unless unexpected inconsistencies are introduced by hardware or software failures. Developers confirmed that this is a problem and should be further investigated.

11 Related Work

Below we compare EXPLODE to file system testing, software model checking, and static bug finding.

File system testing tools. There are many file system testing frameworks that use application interfaces to stress a "live" file system with an adversarial environment. These testing frameworks are less comprehensive than our approach, but they work "out of the box." Thus, there is no reason not to both test a file system and then test with EXPLODE (or vice versa).

Recently, Prabhakaran *et al* [21] studied how file systems handle disk failures and corruption. They developed a testing framework that uses techniques from [25] to infer disk block types and then inject "type-aware" block failure and corruption into file systems. Their results provide motivation for using existing checksumbased file systems (such as Sun's ZFS [32]). While their technique is more precise than random testing, it does not find the crash errors that EXPLODE does, nor is it as systematic. Extending EXPLODE to similarly return garbage on disk reads is trivial.

Software Model Checking. Model checkers have been previously used to find errors in both the design and the implementation of software systems [1, 3, 7, 13, 15, 16, 18, 30]. Two notable examples are Verisoft [13], which systematically explores the interleavings of a concurrent C program, and Java PathFinder [3] which used a specialized virtual machine to check concurrent Java programs by checkpointing states.

The model checking ideas EXPLODE uses — exhausting states, systematic exploration, and choice — are not novel. This paper's conceptual contribution is dramatically reducing the large work factor that plagues traditional model checking. It does so by turning the checking process inside out. It interlaces the control it needs for systematic state exploration *in situ*, throughout the checked system. As far as we know, EXPLODE is the first example of *in situ* model checking. The paper's engineering contribution is building a system that exploits this technique to effectively check large amounts of storage system code with relatively little effort.

Static bug finding. There has been much recent work on static bug finding (e.g., [1,5,8,9,11,12]). Roughly speaking, because dynamic checking runs code, it is limited to just executed paths, but can more effectively check deeper properties implied by the code (e.g., sync() actually commits data to stable storage or crash recovery works). The errors we found would be difficult to get statically. However, we view static analysis as complementary: easy enough to apply that there is no reason not to use it and then use EXPLODE.

12 Conclusion and Future Work

EXPLODE comprehensively checks storage systems by adapting key ideas from model checking in a way that retains their power but discards their intrusiveness. Its interface lets implementors quickly write storage checkers, or simply compose them from existing components. These checkers run on live systems, which means they do not have to emulate either the environment or pieces of the system. As a result, we often have been able to check a new system in minutes. We used EXPLODE to

find serious bugs in a broad range of real, widely-used storage systems, even when we did not have their source code. Every system we checked had bugs. Our gut belief has become that an unchecked system *must* have bugs — if we do not find any we immediately look to see what is wrong with our checker (a similar dynamic arose in our prior static checking work).

The work in this paper can be extended in numerous ways. First, we only checked systems we did not build. While this shows EXPLODE gets good results without a deep understanding of checked code, it also means we barely scratched the surface of what could be checked. In the future we hope to collaborate with system builders to see just how deep EXPLODE can push a valued system.

Second, we only used EXPLODE for bug-finding, but it is equally useful as an end-to-end validation tool (with no bug fixing intended). A storage subsystem implementor can use it to double-check that the environment the subsystem runs in meets its interface contracts and that the implementor did not misunderstand these contracts. Similarly, a user can use it to check that slipping a subsystem into a system breaks nothing. Or use it to pick a working mechanism from a set of alternatives (e.g., if fsync does not work use sync instead).

Finally, we can do many things to improve EXPLODE. Our biggest missed opportunity is that we do nothing clever with states. A big benefit of model checking is perspective: it makes state a first-class concept. Thus it becomes natural to think about checking as a state space search; to focus on hitting states that are most "different" from those already seen; to infer what actions cause "interesting" states to be hit; and to extract the essence of states so that two superficially different ones can be treated as equivalent. We have a long list of such things to add to EXPLODE in the future.

Acknowledgements

We thank Xiaowei Yang, Philip Guo, Daniel Dunbar, Silas Boyd-Wickize, Ben Pfaff, Peter Pawlowski, Mike Houston, Phil Levis for proof-reading. We thank Jane-Ellen Long and Jeff Mogul for help with time management. We especially thank Ken Ashcraft and Cristian Cadar for detailed comments, Jeremy Sugerman for his help in reasoning about the GSX error, and Paul Twohey and Ben Pfaff for help in the initial stages of this project (described in [31]). We thank Martin Abadi (our shepherd) and the anonymous reviewers for their struggles with our opaque submission. This research was supported by National Science Foundation (NSF) CAREER award CNS-0238570-001 and Department of Homeland Security grant FA8750-05-2-0142.

References

- T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In SPIN 2001 Workshop on Model Checking of Software, May 2001.
- [2] Berkeley DB. http://www.sleepycat.com.
- [3] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In IEEE International Conference on Automated Software Engineering, 2000.
- [4] N. Brown. Private communication., Mar. 2005.
- [5] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. Software: Practice and Experience, 30(7):775–802, 2000.
- [6] E. Clarke, O. Grumberg, and D. Peled. Model Checking. MIT Press, 1999.
- [7] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE* 2000, 2000.
- [8] The Coverity software analysis toolset. http://coverity.com.
- [9] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference* on *Programming Language Design and Implementation*, Berlin, Germany, June 2002.
- [10] G. Dunlap, S. T. King, S. Cinar, M. Basrat, and P. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, Dec. 2002.
- [11] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings* of Operating Systems Design and Implementation, Sept. 2000.
- [12] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN* 2002 Conference on Programming Language Design and Implementation, pages 234–245, ACM Press, 2002.
- [13] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In Proceedings of the 24th ACM Symposium on Principles of Programming Languages, 1997.
- [14] HFS and HFS+ utilities. http://darwinsource.opendarwin. org/10.2.6/diskdev_cmds-208.11.
- [15] G. J. Holzmann. The model checker SPIN. Software Engineering, 23(5):279–295, 1997.
- [16] G. J. Holzmann. From code to models. In Proc. 2nd Int. Conf. on Applications of Concurrency to System Design, pages 3–10, Newcastle upon Tyne, 11 K 2001
- [17] M. K. Symbolic Model Checking. Kluwer Academic Publishers, 1993.
- [18] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.
- [19] Linux NFS. http://nfs.sourceforge.net/.
- [20] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks. ACM SIGMOD Conference, pages 109–116, June 1988.
- [21] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, pages 206–220, New York, NY, USA, 2005. ACM Press.
- [22] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. ACM Transactions on Computer Systems, 2(4):277–288, Nov. 1984.
- [23] Sandberg, Goldberg, Kleiman, Walsh, and Lyon. Design and implementation of the Sun network file system, 1985.
- [24] A simple block driver. http://lwn.net/Articles/58719/.
- [25] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In Second USENIX Conference on File and Storage Technologies, 2003.
- [26] Linux software RAID. http://cgi.cse.unsw.edu.au/~neilb/ SoftRaid.
- [27] Subversion. http://subversion.tigris.org.
- [28] J. Sugerman. Private communication., Dec. 2005.
- [29] VMware GSX server. http://www.vmware.com/products/ server/.
- [30] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In Proceedings of the Sixth Symposium on Operating Systems Design and Implementation, Dec. 2004.
- [31] J. Yang, P. Twohey, B. Pfaff, C. Sar, and D. Engler. eXplode: A lightweight, general approach for finding serious errors in storage systems. In Workshop on the Evaluation of Software Defect Detection Tools, June 2005.
- [32] Zfs: the last word in file systems. http://www.sun.com/ 2004-0914/feature/.

Securing software by enforcing data-flow integrity

Miguel Castro Microsoft Research Manuel Costa Microsoft Research University of Cambridge Tim Harris Microsoft Research

Abstract

Software attacks often subvert the intended data-flow in a vulnerable program. For example, attackers exploit buffer overflows and format string vulnerabilities to write data to unintended locations. We present a simple technique that prevents these attacks by enforcing data-flow integrity. It computes a data-flow graph using static analysis, and it instruments the program to ensure that the flow of data at runtime is allowed by the data-flow graph. We describe an efficient implementation of data-flow integrity enforcement that uses static analysis to reduce instrumentation overhead. This implementation can be used in practice to detect a broad class of attacks and errors because it can be applied automatically to C and C++ programs without modifications, it does not have false positives, and it has low overhead.

1 Introduction

Most software is written in unsafe languages like C and C++. Even programs written in type-safe languages have libraries and runtimes written in unsafe languages. Therefore, software is vulnerable to attacks and it is likely to remain vulnerable in the foreseeable future.

Almost all these attacks subvert the intended data-flow in the program. They exploit software vulnerabilities to write data to unintended locations. For example, control-data attacks exploit buffer overflows or other vulnerabilities to overwrite a return address [32], a function pointer [28], or some other piece of control-data. Non-control-data attacks exploit similar vulnerabilities to overwrite security critical data without subverting the intended control-flow in the program [14]. Non-control-data attacks have not been observed in the wild but they are just as serious and there are no good defenses against them. Non-control-data attacks will become common as we deploy defenses for control-data attacks.

This paper presents a technique that can prevent both

control and non-control-data attacks by enforcing a simple safety property that we call *data-flow integrity*. This technique computes a data-flow graph for a vulnerable program using static analysis, and instruments the program to ensure that the flow of data at runtime is allowed by the data-flow graph. It can be applied to existing C and C++ programs automatically because it requires no modifications and it does not generate false positives.

There are many proposals to prevent attacks on software, for example, [30, 24, 27, 5, 18, 34, 31, 16, 19, 13, 37]. CCured [30] and Cyclone [24] propose memorysafe dialects of C that prevent all these attacks. The disadvantage of these approaches is that the effort to port existing C code to these dialects is non-trivial and they require significant changes to the C runtime, for example, they replace malloc and free by a garbage collector. There are several techniques that can be applied to existing programs but can only defend from attacks that overwrite specific targets, for example, return addresses [18], or that exploit specific types of vulnerabilities, for example, buffer overflows [25, 34]. Program shepherding [27] and control-flow integrity [5] provide a generic defense against control-data attacks but they cannot defend against non-control-data attacks. Techniques that perform dynamic taint analysis [37, 15, 19, 31, 13, 16, 22, 33] can prevent control-data attacks and they can prevent some non-control-data attacks [37, 16, 13], but they may have false positives and they incur a very high overhead without hardware support.

We implemented data-flow integrity enforcement using the Phoenix compiler infrastructure [29]. The implementation uses reaching definitions analysis [7] to compute a static data-flow graph. For each value read by an instruction, it computes the set of instructions that may write the value. The analysis relies on the same assumptions that existing compilers rely on to implement standard optimizations. These are precisely the assumptions that attacks violate and data-flow integrity enforcement detects when they are violated.

To enforce data-flow integrity at runtime, our implementation instruments the program to compute the definition that actually reaches each use at runtime. It maintains a table with the identifier of the last instruction to write to each memory position. The program is instrumented to update this table before every write and to prevent the attacker from tampering with the table. We also instrument reads to check if the identifier of the instruction that wrote the value being read is an element of the set computed by the static analysis. If it is not, we raise an exception. Our implementation does not generate false positives; when we raise an exception, the program has an error.

We have developed a number of optimizations to reduce the instrumentation overhead. The first optimization computes equivalence classes of instructions and assigns the same identifier to all the instructions in the same class. This reduces the number of bits required to represent identifiers and simplifies the code to check set membership on reads. Additionally, we perform static analysis of the low level intermediate representation of the compiler to remove unnecessary read and write instrumentation. This analysis is more conservative than the one used to compute the data-flow graph; it does not rely on any assumptions that may be invalidated by attacks.

We evaluated the efficacy and overhead of our implementation. The results show that data-flow integrity enforcement can prevent many control-data and noncontrol-data attacks, and that it can detect errors in existing programs. The instrumentation overhead is low: the space overhead is approximately 50%, and the average runtime overhead is between 44% and 103% in CPU intensive Spec 2000 benchmarks. In a Web server running Spec Web 1999, the runtime overhead is even lower: the average response time increases by 0.1% and peak throughput decreases by 23%. Thus, data-flow integrity enforcement can be used in practice to defend software from attacks.

2 Data flow integrity enforcement

This section starts by providing a high level overview of data-flow integrity enforcement. Then it describes in detail the static analysis and the instrumentation.

2.1 Overview

Data-flow integrity enforcement has three phases. The first phase uses static analysis to compute a data-flow graph for the vulnerable program. The second instruments the program to ensure that the data-flow at runtime is allowed by this graph. The last one runs the instrumented program and raises an exception if data-flow

integrity is violated. We will use the simple example in Figure 1 to illustrate how these phases work.

```
1: int authenticated = 0;
2: char packet[1000];
3:
4: while (!authenticated) {
5:
     PacketRead(packet);
6:
7:
     if (Authenticate(packet))
8:
       authenticated = 1;
9: }
10:
11: if (authenticated)
12:
      ProcessPacket (packet);
```

Figure 1: Example vulnerable code in C.

Figure 1 shows a code fragment that is inspired by a vulnerability in SSH [3] that can be exploited to launch both control-data and non-control-data attacks [14]. In this example, we assume that PacketRead can write more than 1000 bytes to packet when it receives a packet from the network. This vulnerability could be exploited to overwrite the return address of the function or to overwrite authenticated. The first attack is a form of control-data attack that may allow the attacker to gain control over the execution. The second is a non-control-data attack that allows the attacker to bypass authentication and get its packet processed. Data-flow integrity enforcement can prevent both attacks.

We use reaching definitions analysis [7] to compute the static data-flow graph. Using the terminology from [7], an instruction that writes to a memory position *defines* the value in the memory position, and an instruction that reads the value is said to *use* the value. The analysis computes the set of reaching definitions for each use and assigns an identifier to each definition. It returns a map from instructions to definition identifiers and a set of reaching definition identifiers for each use, which we call the static data-flow graph.

For example, authenticated is used in lines 4 and 11. If we ran reaching definitions analysis in the source code, it might conclude that the definitions in lines 1 and 8 reach both uses. Therefore, the set of reaching definition identifiers for both uses would be $\{1,8\}$, if we used the line numbers to identify the definitions.

The analysis can be imprecise but it is important that it be conservative. It must include in the set all definitions that may reach a use at runtime but it may include additional definitions. For example, only the definition in line 8 can reach the use in line 11 but the analysis might compute the set of reaching definitions $\{1,8\}$. This ensures that imprecisions can lead to false negatives but not false positives: data-flow integrity enforcement may miss

some attacks but it will never signal an error unless there is one. We believe that this property is very important because users are not likely to deploy security solutions that can break running systems in the absence of errors.

The second phase instruments the program to enforce a simple safety property that we call *data-flow integrity*, i.e., whenever a value is read, the definition identifier of the instruction that wrote the value is in the set of reaching definitions for the read (if there is one).

The program is instrumented to compute the definition that reaches each read at runtime and to check if this definition is in the set of reaching definition identifiers that was computed statically. To compute reaching definitions at runtime, we maintain the *runtime definitions table* (RDT) that records the identifier of the last instruction to write to each memory position. Every write is instrumented to update the RDT. The instrumentation before reads uses the address of the value being read to retrieve the identifier from the RDT. Then, it checks if this identifier is in the statically-computed set. For example in Figure 1, we would add code to set RDT[&authenticated] to 8 in line 8, and to check if RDT[&authenticated] $\in \{1,8\}$ in lines 4 and 11.

We want to enforce data-flow integrity in the presence of a strong attacker that can write anywhere and that can even execute data. To achieve this goal, the attacker must be prevented from tampering with the RDT, tampering with the code, or bypassing the instrumentation.

RDT tampering is prevented by instrumenting writes to check if the target address is within the memory region allocated to the RDT. Any attempt to write to the RDT generates an exception. Code tampering can be prevented with the same checks or by using read-only protection for code pages; we use the latter, which is already available in most processors.

To prevent the attacker from bypassing the instrumentation, we must prevent tampering with the target addresses of indirect control transfers. We instrument writes and reads of programmer-defined control-data as described before. In addition, we instrument writes and reads of control-data added by the compiler in the same way. For example, we set the RDT entry for a return address to a well-known value and we check if the entry still holds the value on return. In our test cases, this instrumentation is sufficient to prevent the attacker from tampering with the target addresses of indirect control transfers. However, there may be cases where the reaching definitions analysis is not precise enough. For example, the reaching definitions set for the read of a function pointer may contain the identifier of an array store instruction that an attacker can use to overwrite the function pointer. We can detect when this happens and use existing techniques to ensure control-flow integrity [27, 5].

It is possible to trade off coverage for lower overhead by removing reaching definitions sets for some uses: some attacks may go undetected because those uses are not instrumented but the overhead will be lower. We can still ensure data-flow integrity for all other uses provided uses of control-data are instrumented. We experimented with a variant of our implementation that only instruments uses of control-data and uses of local variables without definitions external to the function. This variant is interesting because it has low overhead and it can still catch many interesting attacks. For example, it can prevent any attack that violates control-flow integrity and the attack in our example.

When the instrumented program runs, any deviation from the data-flow graph computed statically raises an exception. Since the analysis is conservative, there are no false positives. If there is an exception, the program has an error. The error may or not be triggered by an attack. In addition to detecting various known attacks, we found several unknown errors in our test cases.

2.2 Static analysis

We compute reaching definitions using a combination of two analyses: a flow-sensitive intra-procedural analysis and a flow-insensitive and context-insensitive interprocedural analysis. We implemented both analyses using the Phoenix compiler framework [29]. They operate on Phoenix's high level intermediate representation (HIR), which enables them to be applied to different languages and different target architectures. Figure 2 shows the HIR for the vulnerable C code in Figure 1.

The intra-procedural analysis takes flow control into account. It is implemented by traversing the static single-assignment representation [10] produced by Phoenix. We use this analysis to compute reaching definitions for uses of local variables that have no definitions external to the function in which they are declared. The inter-

```
_authenticated = ASSIGN 0
                                              #1
$L6: t274 = CMP(NE) _authenticated, 0
                                              #2
              CBRANCH (NE) t274, $L7, $L8
                                              #3
      tv275 = CALL &_PacketRead, &_packet
$L8:
      t276 = CALL &_Authenticate, &_packet
                                              #5
      t277 = CMP(NE) t276, 0
                                              #6
              CBRANCH (NE) t277, $L10, $L9
                                              #7
$L10: _authenticated = ASSIGN 1
                                              #8
$L9:
             GOTO $L6
                                              #9
$L7: t278 = CMP(NE) _authenticated, 0
                                              #10
              CBRANCH (NE) t278, $L12, $L11
                                              #11
$L12: tv279 = CALL &_ProcessPacket, &_packet #12
$1.11:
```

Figure 2: Example vulnerable code in high-level intermediate representation (HIR).

procedural analysis is used to compute reaching definitions for all other uses.

The inter-procedural analysis is less precise to allow it to scale to large programs. It ignores control-flow and it does not take the calling context into account when analyzing functions. We implemented Andersen's points-to analysis [9] to compute the set of objects that each pointer can point to, and we use these points-to sets to compute reaching definitions. The implementation is similar to the one described in [21] but it is field-insensitive rather than field-based (i.e., it does not distinguish between the different fields in a structure, union, or class). These imprecisions can lead to false negatives, for example, we may fail to detect an attack that overflows a buffer in a structure to overwrite a security critical field in the same structure. We plan to implement an analysis with better precision in the future (e.g., [12]).

The points-to analysis makes a global pass over all source files to collect *subset constraints*. Each assignment x = y results in a subset constraint $x \supseteq y$, which means that the set of possible values of x contains the set of possible values of y. The analysis uses Phoenix to compile each source file to HIR and it writes all subset constraints in the HIR to a file. After this global pass, it computes the points-to sets by iterating over all the constraints until it reaches a fixed point. Then, it stores the points-to sets in a file.

During the global pass, we also collect the target locations and identifiers of instructions that write to locations that may be read in other functions. These include writes to locations obtained by dereferencing pointers, to static and global variables, and to local variables whose address is taken. This information is also written to a file.

We compute inter-procedural reaching definitions using the points-to sets and information about write instructions collected during the global pass. For uses of variables and temporaries, the set of reaching definitions is the union of the set containing the identifiers of all writes to the variable (or temporary) with the sets containing the identifiers of all writes to dereferences of pointers that may point to the variable (or temporary). For pointer dereferences, the set of reaching definitions is the union of the set containing the identifiers of all writes to the dereferenced pointer with the sets of reaching definitions of all the variables the pointer can point to. The sets of inter-procedural reaching definitions are written to a file that is used to instrument the program.

Both the intra-procedural and the inter-procedural analyses assume that the relative layout of independent objects in memory is undefined [12]. They assume that correct programs do not use pointer arithmetic to navigate between independent objects in memory. For example in Figure 2, the analyses assume that correct programs will not use a pointer to the packet array to write

to the authenticated variable. Existing compilers already make this assumption when implementing several standard optimizations. Therefore, this assumption applies to the vast majority of programs. However, it is precisely this assumption that is violated by most attacks. Data-flow integrity enforcement detects and prevents these attacks.

We applied the analysis to the sample code in Figure 2 using the numbers at the end of the lines to identify definitions. The set of reaching definitions is {1,8} for both uses of authenticated (in lines 2 and 10). The temporaries have a single definition in the preceding instruction. Since we control code generation, we can ensure that these temporaries are placed in registers beyond the reach of an attacker. The attacker cannot violate dataflow integrity by overwriting these registers because our instrumentation prevents it from subverting the control flow. We only compute reaching definitions and instrument accesses of temporaries that are spilled to memory.

2.3 Instrumentation

We add instrumentation by inserting new high-level instructions into the HIR of the program. The instructions have the form:

SETDEF opnd id CHECKDEF opnd setName.

The first instruction sets the RDT entry for opnd to id. The second retrieves the runtime definition identifier for opnd from the RDT and checks if the identifier is in the reaching definitions set with name setName. The compiler maintains a map from set names to set values that is used when lowering CHECKDEF instructions to the assembly of the target machine. Instrumenting a high-level representation of the code has the advantage of making the instrumentation machinery independent of the source language and mostly independent of the target architecture. Currently, we only target the x86 architecture but it would be simple to target other architectures.

Figure 3 shows the HIR for the vulnerable code with high-level instrumentation generated from the information computed by the reaching definitions analysis. The set with name 100 has the value {1,8}. We do not instrument temporaries that we can ensure are allocated to registers, and we also do not instrument the uses of &packet because addresses of local variables are computed by adding a constant to the frame pointer.

Before describing how the high-level instrumentation is lowered to assembly, we need to describe how we implement the RDT. To enable efficient accesses, the RDT is implemented as an array with a definition identifier for each 32-bit memory word in the instrumented program. Each definition identifier is two bytes long, which seems sufficient even for large programs. This results in a space

```
SETDEF _authenticated 1
                                             #1
     _authenticated = ASSIGN 0
$L6: CHECKDEF _authenticated 100
      t274 = CMP(NE) _authenticated, 0
                                             #2
             CBRANCH(NE) t274, $L7, $L8
                                             #3
$L8: tv275 = CALL &_PacketRead, &_packet
                                             #4
     t276 = CALL &_Authenticate, &_packet
                                             #5
     t277 = CMP(NE) t276, 0
                                             #6
             CBRANCH(NE) t277, $L10, $L9
                                             #7
$L10: SETDEF _authenticated 8
     _authenticated = ASSIGN 1
                                             #8
$L9:
             GOTO $L6
                                             #9
$L7: CHECKDEF _authenticated 100
     t278 = CMP(NE) _authenticated, 0
                                             #10
             CBRANCH(NE) t278, $L12, $L11
                                             #11
$L12: tv279 = CALL &_ProcessPacket, &_packet #12
```

Figure 3: Example vulnerable code in HIR with instrumentation.

overhead of approximately 50%.

There are some subtle issues on the choice of memory granularity for recording definition identifiers. Since programs can access memory at byte granularity, it would seem necessary to record a 2-byte definition identifier for every byte of memory. This would result in a space overhead of approximately 200%, which may not be practical. We are able to record a single identifier for each 32-bit word because we can generate code in which no two variables with distinct reaching definition sets share the same aligned 32-bit memory word. Since our reaching definitions analysis does not distinguish between different fields in objects and between different elements in arrays, it is not necessary to change the layout of arrays and objects. We only changed the compiler to use a minimum alignment of 32 bits when laying out local variables in a stack frame and globals in the data segment. Function arguments and heap allocated objects were already appropriately aligned.

In the current implementation, we allocate the lowest 1GB of the virtual address space to the program being instrumented and 512MB to the RDT with a guard page between them, that is, the guard page is at address 40000000h and the base of the RDT is at address 40001000h. So, to compute the address of the RDT entry for an operand, we simply take the address of the operand shift it right by two, multiply the result by two, and add 40001000h. This layout also enables efficient bounds checking of the target addresses of writes to prevent tampering with the RDT: we raise an exception if the bitwise and of the target address with c00000000h is non-zero (as in [39]). The guard page allows us to check only the target address for the write and ignore the size.

The high-level instrumentation is lowered to x86 assembly as illustrated by the following examples. We

lower SETDEF authenticated 1 to:

```
lea ecx,[_authenticated]
test ecx,0C0000000h
je L
int 3
L: shr ecx,2
mov word ptr [ecx*2+40001000h],1
```

The first instruction loads the target address of the write into ecx and the following three instructions perform the bounds check on the address. If the check fails, we currently generate a breakpoint (int 3), which is very convenient for debugging. Another exception would be more appropriate in production use. The shr instruction is used to compute the address of the RDT entry for _authenticated and the mov instruction updates the entry. If the size of the operand is greater than 32 bits, it is necessary to update the entries in the RDT corresponding to the other words. We can update entries for 64-bit operands with a single mov instruction by moving the concatenation of two copies of the identifier. But we add additional instructions with larger operands.

The CHECKDEF authenticated 100 instruction is lowered to:

```
lea ecx,[_authenticated]
shr ecx,2
mov cx, word ptr [ecx*2+40001000h]
cmp cx,1
je L
cmp cx,8
je L
int 3
L:
```

This code compares the definition identifier in the RDT entry for _authenticated with the definition identifiers in set 100. When the operand is larger than 32 bits, we add additional comparisons for the other words.

In addition, we instrument definitions and uses of control-data introduced in the compilation process. For example on function entry, we add the following code to set the RDT entry corresponding to the function's return address to zero:

```
mov ecx, esp
shr ecx,2
mov word ptr [ecx*2+40001000h],0
```

Just before returning, we add code to check if the definition identifier that reaches the use of the return address is zero.

```
mov ecx, esp
shr ecx,2
cmp word ptr [ecx*2+40001000h],0
```

```
je L
int 3
L: ret
```

Going back to our example, the instrumented code is no longer vulnerable to either the control-data attack that overwrites the return address or the non-control-data attack that overwrites authenticated. Since the analysis concludes that authenticated is not aliased with packet, writes to packet have identifiers that are guaranteed to be distinct from 1 or 8. Additionally, the identifier zero is only used on function entry for return addresses. Therefore, any return address overwrite would also be detected.

2.4 Runtime environment

Programs rely on a complex runtime environment that includes several libraries. It is often impossible to analyze the source code of these libraries. Frequently, only the binaries are available and, even when source code is available, some functions are hand-written in assembly. Yet, many attacks make use of libraries when exploiting vulnerabilities. For example, string manipulation functions in the C library are notorious for their use in exploits of buffer overflow vulnerabilities.

Techniques that use source code analysis to instrument writes fail to provide any guarantees unless library calls are wrapped to perform safety checks. These techniques include array bounds checking [34, 25] and memory-safe dialects of C [30, 24]. Sometimes wrappers are also required to perform memory layout conversions [30]. Writing these wrappers can be onerous.

Data-flow integrity enforcement does not require any wrappers for library functions that are not analyzed. If a programs calls these functions, we cannot instrument some uses but we guarantee integrity of the data-flow for all other uses. To do this, we instrument library binaries to set the RDT entries for any memory they write to an invalid definition identifier (which can be done without the source code). To prevent false positives, we do not instrument uses of memory that is reachable from a pointer that is passed to or received from one of these library functions. We determine these uses by running a simple reachability analysis on the output of our pointsto analysis. We instrument all other uses. In particular, we can always instrument all uses of local variables without definitions external to the function where they are declared, which prevents the attack in our example.

We provide the option to define library wrappers to increase coverage. To define a wrapper for a library function, one must write a wrapper function and describe the subset constraints that calling the function adds to the points-to analysis. The wrapper function checks definition identifiers for the memory read by the library, calls

the library, and sets definition identifiers for the memory written by the library. We instrument the code to call the wrapper instead of the original function and to supply the wrapper with reaching definition sets for the memory read by the library function and a definition identifier for the memory it writes. For example, a wrapper for the Windows NT operating system call CreateProcess can check the integrity of the application name and command line strings supplied as arguments.

Figure 4: Example library function wrapper.

We have written wrappers for library functions used in our test cases, which include some operating system calls. Figure 4 shows an example wrapper for memcpy. CHECK_BOUNDS ensures that memcpy does not write into the RDT and CHECK_ARRAY checks if the identifiers in the RDT for the bytes in src are in the reaching definitions set supplied in defArgs. The RDT entries for the bytes written to dest are set to defId by UPDATE_RDT.

3 Optimizations

A naïve implementation of data-flow integrity enforcement can perform poorly: each definition introduces a write to the RDT and each use check introduces a read from the RDT followed by comparisons against each identifier in the set of reaching definitions for the use. This section discusses a number of techniques that we have developed to reduce this overhead.

3.1 Renaming equivalent definitions

The first optimization partitions definitions into equivalence classes in a way that allows us to safely assign the same identifier to all definitions in the same class. Two definitions are *equivalent* if they have exactly the same set of uses. This reduces the number of comparisons in CHECKDEFs and the number of bits required to represent identifiers. For example, both definitions of _authenticated in Figure 2 have the same set of uses

computed by the static analysis. We assign the same identifier 1 to both definitions. Therefore, CHECKDEF authenticated 100 requires only one comparison. It is compiled to:

```
lea ecx,[_authenticated]
shr ecx,2
cmp word ptr [ecx*2+40001000h],1
je L
int 3
L:
```

Our experiments show that this optimization is fundamental to obtain low overhead.

3.2 Removing bounds checks on writes

We check the target addresses of writes to prevent the attacker from tampering with the RDT. We can optimize SETDEFs by removing these checks from all *safe* writes. In the current implementation, a write is safe if the target address is obtained by adding a small constant offset (possibly zero) to the stack pointer, frame pointer, or to the address of a global or static variable. We require the sum of the offset and the size of the data being written to be less than 4KB (which is the size of the guard page that we allocate before the RDT).

For example in Figure 3, since _authenticated is a local variable whose address is obtained by adding a small constant to the frame pointer, we can remove the bounds check from SETDEF authenticated 1. The SETDEF is compiled to:

```
lea ecx,[_authenticated]
shr ecx,2
mov word ptr [ecx*2+40001000h],1
```

3.3 Removing SETDEFs and CHECKDEFS

The next optimization runs data-flow analysis to remove some SETDEFs and CHECKDEFs. We must be careful to avoid two problems. First, we must not rely on high level analyses whose inferences are unsound once data-flow integrity has been lost: the entire purpose of the instrumentation is to detect cases where the program's data-flow integrity is compromised. Second, we must not remove checks too early during compilation because later code transformations may change the situations in which data-flow integrity is lost. Therefore, we perform our optimizations when SETDEF and CHECKDEF operations are still present in their HIR form but the remainder of the program has already been lowered to the native instruction set and is ready to emit.

For simplicity, we will describe the case of instruction sequences within the same basic block. Our implementation deals with the case of propagating flow variables into a basic block if it has exactly one predecessor. It could be extended to provide more optimization opportunities by using the techniques in [8, 20].

Our instrumentation is redundant in the following cases. Suppose that instructions I_1 and I_2 are a pair of SETDEFS or CHECKDEFS relating to the same data that execute without any intervening write to that data.

- 1. If I_1 and I_2 are both SETDEFs with the same identifier then I_2 is redundant.
- 2. If I_1 and I_2 are both SETDEFs with no intervening CHECKDEF for that data then I_1 is redundant.
- 3. If I_1 is a SETDEF for ID_1 and I_2 a CHECKDEF for a set containing ID_1 then I_2 is redundant (indeed, ID_1 must be in I_2 's set if the data-flow analysis was performed correctly).
- 4. If I_1 and I_2 are both CHECKDEFs against sets IDS_1 and IDS_2 respectively then IDS_2 can be reduced to contain only elements in IDS_1 (the earlier check guarantees no other elements are present). Furthermore, if IDS_1 and IDS_2 hold identical elements then I_2 can be removed (it is not possible for the latter check to fail if the earlier check succeeds).
- 5. If I_1 is a CHECKDEF against set IDS_1 and I_2 a SETDEF for ID_2 then I_2 is redundant if $IDS_1 = \{ID_2\}$.

In practice rules 3 and 4 are the most effective. Rule 3 eliminates many CHECKDEF instructions when uses of data occur close to their definitions. Rule 4 lets us remove CHECKDEF instructions where the same data is used repeatedly, even if there are aliasing writes between the data's definition and the first of those uses.

To identify redundant instrumentation, we use symbolic execution of the native code augmented with SETDEF and CHECKDEF operations. We update the symbolic state of the registers after each instruction and the symbolic state of the RDT after SETDEFs and CHECKDEFs. The symbolic state of the RDT maps symbolic memory addresses to sets of definition identifiers.

The current implementation uses a simple test to compare symbolic addresses. Two addresses are equal if they are syntactically equal. They are different if they are computed by adding different offsets to the same symbolic register state. Otherwise, they may refer to aliased memory locations. A write to memory invalidates the symbolic state of a register if this state refers to the contents of a memory position that may be aliased with the write's target. Additionally, it removes mappings for any memory that may be aliased with the write's target from the symbolic RDT state. We apply the rules to eliminate redundant instrumentation after each SETDEF and CHECKDEF by examining the symbolic RDT state.

3.4 Optimizing membership checks

Another optimization renames definitions to reduce the cost of membership checks in CHECKDEFs. Membership checks can be implemented more efficiently when sets contain ranges of consecutive identifiers: a check against $\{0 \dots n\}$ can be implemented by a single unsigned integer comparison against n, and a check against $\{n \dots m\}$ can be implemented by subtracting n and performing an unsigned comparison against m-n.

We define the cost of a CHECKDEF as the number of subtractions and comparisons necessary to perform its membership check. For instance, the cost of checking membership in $\{1,3,7\}$ is three, but it is only one for $\{0,1,2\}$. We say that the $total\ cost$ of an identifier set is the cost of a single CHECKDEF against it, multiplied by the number of CHECKDEFs against it that occur in the program text.

We use a simple greedy algorithm to attempt to reduce the sum of the total costs of all sets: we sort the sets in order of decreasing total cost and proceed to assign the most costly sets to contiguous identifier ranges. We start from identifier 0 and so the set with the largest total cost benefits from the cheapest comparisons. Once we have started assigning identifiers to the elements of popular sets, we are constrained in the assignment of identifiers to elements in less popular sets that intersect them.

3.5 Removing SETDEFs for safe definitions

The last optimization identifies local variables that have no definitions outside the function and that are written only by safe writes (according to the definition of safe in 3.2). It replaces all SETDEFs for such a variable by a single SETDEF with identifier 0 that is placed on function entry. It also simplifies CHECKDEFs for such variables to have a reaching definitions set of $\{0\}$. This is sufficient because it detects violations of data-flow integrity that overwrite these variables and safe writes cannot overwrite other variables.

3.6 Other optimizations

We also tried to reorder comparisons in CHECKDEFS to take advantage of the fact that different definitions can reach a use with different frequencies at runtime. However, this optimization had little impact in our sample programs after we applied the other optimizations. Therefore, it is not included in the results that we present.

We have focused on optimizations that can be made without reference to profiling data. Using feedback, whether dynamically or from full program runs, is an interesting direction for future work.

4 Evaluation

We ran experiments to evaluate the overhead of our implementation and the effectiveness of data-flow integrity enforcement at preventing control-data and non-controldata attacks. This section presents our results.

4.1 Overhead

We used several programs from the SPEC CPU 2000 benchmark suite to measure the overhead added by our instrumentation. We chose these programs to facilitate comparison with other techniques that have been evaluated using the same benchmark suite, for example, [5]. We ran six integer benchmarks (gzip, vpr, mcf, crafty, bzip2, and twolf) and three floating point benchmarks (art, equake, and ammp).

These benchmarks are CPU-intensive and they spend most time executing instrumented code at user level. The overhead of our instrumentation is likely to be higher in these benchmarks than in other programs where it would be masked by other overheads. Therefore, we also measured the overhead in a Web server running the SPEC WEB 1999 benchmark (see Section 4.1.4).

We compared the running time and peak physical memory usage of the programs compiled using Phoenix with and without our instrumentation. The version without instrumentation is compiled with optimization (/Ox) but the current release of Phoenix does not implement many optimizations that are available in other compilers.

We ran the experiments on an idle Dell Precision Workstation 350 with a 3GHz Intel Pentium 4 processor and 2GB of memory. The operating system was Windows XP professional with service pack 2. We ran each experiment three times and we present the average result. The variance in the results was negligible.

4.1.1 Comparison with uninstrumented code

The first set of experiments measured the overhead of data-flow integrity enforcement with all the optimizations. We evaluated two variants of data-flow integrity enforcement: *intraproc DFI* only instruments uses of control-data and uses of local variables without definitions outside their function, and *interproc DFI* is the variant described earlier in the paper.

Figure 5 shows the execution time of the two variants of data-flow integrity enforcement normalized by the execution time without instrumentation (which is labeled *base* in the figure). The average overhead is 43% for *intraproc DFI* and 104% for *interproc DFI*. We believe that the overhead of either variant is sufficiently low to enable the use of data-flow integrity enforcement in applications that are not CPU intensive or that spend more

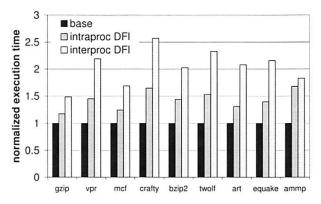


Figure 5: Execution time with data-flow integrity enforcement (normalized by the execution time without instrumentation).

time executing in the kernel. For example, it could be applied to Internet browsers or Web servers, which are the target of frequent attacks. In Section 4.1.4, we show that the overhead is significantly lower when using data-flow integrity enforcement to protect a Web server.

It is hard to perform a detailed quantitative comparison with previous techniques due to differences in hardware, operating system, and compiler. But we can use published results obtained using the same benchmarks to put our overhead in perspective. Program Shepherding [27] and CFI [5] have lower overhead but these techniques cannot detect non-control-data attacks. The overhead of either variant of DFI is significantly lower than the overhead incurred by a state-of-the-art C bounds checker: CRED [34] incurs an overhead of nearly 300% in bzip2 and 100% in gzip [34]. The overhead of software implementations of taint checking [31, 16] is also significantly higher, for example, TaintCheck [31] ran bzip2 37.2 times slower than without instrumentation.

We also compared the space overhead introduced by *interproc DFI* relative to *base* during the execution of each program. We compare the peak physical memory usage during execution, as reported by the operating system. The results are shown in Figure 6. As expected the overhead is approximately 50% because we allocate a 2-byte entry in the RDT for each 4-byte word in the instrumented program.

4.1.2 Overhead breakdown

To better understand the sources of overhead, we ran experiments to breakdown the overhead of *interproc DFI* into several components. The experiments removed components from the instrumentation one at a time. Figure 7 shows the breakdown normalized by the total overhead for each program.

The overhead of SETDEFs is the sum of the overhead

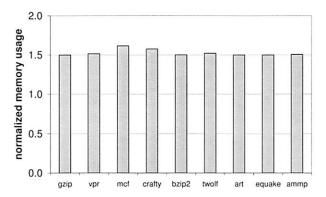


Figure 6: Space overhead of data-flow integrity enforcement relative to the execution without instrumentation.

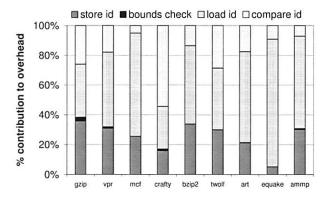


Figure 7: Breakdown of the instrumentation overhead.

of the first two components: storing the identifier into the RDT and performing a bounds check to protect the RDT from tampering. The remaining overhead is due to CHECKDEFS: loading the identifier from the RDT and checking if it is in the set of reaching definitions. A SETDEF is on average 3.6 times more expensive than a CHECKDEF but the programs execute between 4 times and 25 times more CHECKDEFS than SETDEFS. Therefore, the total overhead of CHECKDEFS is significantly higher than that of SETDEFS.

The bulk of the overhead is due to memory accesses to the RDT and the cache pollution they introduce, which is not surprising given the gap between processor speed and memory latency. The overhead of the bounds check on writes is negligible because our optimization can remove many bounds checks. Membership checks are more expensive because we always perform at least one comparison and sometimes more: the average number of comparisons is 1.5. These membership checks account for most of the overhead in crafty because its small working set makes accesses to the RDT relatively less expensive.

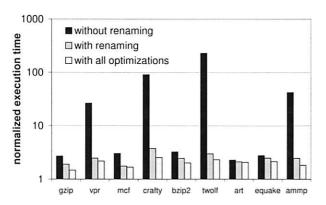


Figure 8: Execution time without optimizations, with the renaming optimization, and with all optimizations (normalized by the execution time without instrumentation).

4.1.3 Impact of optimizations

We also measured the overhead reduction afforded by the different optimizations.

The optimization with the most significant impact is the renaming optimization, which assigns the same identifier to all definitions that have the same set of uses. We compared the running time of *interproc DFI* with and without this optimization. The version without the optimization assigns a unique identifier to each instruction that defines a value. Figure 8 shows the execution times without optimizations, with the renaming optimization, and with all the optimizations. The execution times are normalized by the execution time without instrumentation and the y-axis has a logarithmic scale.

The results show that the renaming optimization is fundamental for data-flow integrity enforcement to be practical: some benchmarks are more than an order of magnitude slower without it. The performance difference is due to a large reduction in the size of reaching definition sets. We measured the size of reaching definition sets computed by the interprocedural analysis with and without the optimization. Figure 9 shows the median and 90-th percentile set sizes without the optimization divided by the corresponding values with the optimization. The ratios are very large: they vary between 3 and 264. The renaming optimization brings the median definition set size to 3 or less for 6 of the benchmarks.

We also ran experiments to measure the impact of the other optimizations discussed in Section 3. The experiments added one optimization at a time starting from the version with the renaming optimization. Figure 10 shows the contribution of each optimization to the difference in execution time between the version with the renaming optimization and the version with all optimizations.

The results show that all the optimizations play an important role in at least one benchmark program.

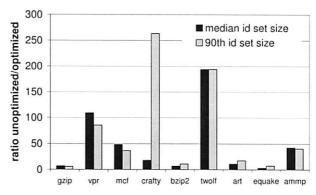


Figure 9: Median and 90-th percentile definition set sizes without the renaming optimization divided by the corresponding values with the optimization.

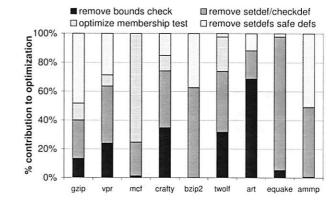


Figure 10: Contribution of each optimization (other than renaming) to the reduction in execution time.

4.1.4 Web server overhead

We also ran the SPEC Web 1999 benchmark to measure the overhead added by our instrumentation to the NullHttpd Web server. The server ran on a Dell Precision workstation 350 with a 3GHz Intel Pentium 4 processor and 2GB of RAM. The clients ran on a Dell Latitude D600 laptop with a 2GHz Intel Pentium processor and 1GB of RAM. Both machines ran Windows XP professional with service pack 2 and they were connected by a 100Mbps D-Link Ethernet switch. We configured the clients to request only static content. Since we do not instrument the Perl scripts that handle requests for dynamic content, our overhead is higher with static content. We present the average of three runs.

We measured the average response time and throughput for a base version of the server without instrumentation and for a version of the server instrumented to enforce data-flow integrity. For both versions, we increased the number of clients while the results complied with the benchmark rules. The results stopped complying at 60 clients for the version with instrumentation and at 80 clients for the version without instrumentation. Figure 11

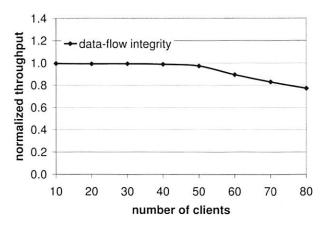


Figure 11: Spec Web throughput with instrumentation normalized by the throughput without instrumentation.

shows the throughput in operations per second for the version with instrumentation normalized by the throughput without instrumentation.

When server load is low, the overhead of our instrumentation is also low because it is masked by other overheads (for example, the time to send requests and replies across the network). The results show that the throughput of both versions of the server is almost identical with up to 40 clients. Additionally, the average operation response time in an unloaded server (10 clients) is only 0.1% longer with instrumentation than without.

When server load is high, the overhead of data-flow integrity enforcement increases because the server is CPU-bound in this benchmark. The overhead of enforcing data-flow integrity increases up to a maximum of 23%.

4.2 Effectiveness against attacks

We used several exploits to evaluate the effectiveness of data-flow integrity enforcement at stopping both controldata and non-control-data attacks. We used a benchmark with synthetic exploits [40] and several exploits of real vulnerabilities in existing programs. This section describes the programs, the vulnerabilities, and the exploits.

4.2.1 Synthetic exploits

We ran the benchmark program that was described in [40]. This benchmark has 18 control-data attacks that exploit buffer overflow vulnerabilities. The attacks are classified according to the technique they use to overwrite control-data, the location of the buffer they overflow, and the control-data they target. There are two techniques to overwrite control-data. The first overflows a buffer until the control-data is overwritten. The second overflows a buffer until a pointer is overwritten, and

uses an assignment through the pointer to overwrite the control-data. The attacks can overflow buffers located in the stack or in the data segment, and they can target four types of control-data: the return address on the stack, the old base pointer on the stack, and function pointers and longimp buffers in the stack or in the data segment.

Table 1 shows that data-flow integrity enforcement can prevent all the attacks in the benchmark. The best techniques evaluated in [40] failed to prevent 50% of the attacks. Some recent techniques can also prevent all attacks in this benchmark, for example, CFI [5] and CRED [34].

4.2.2 Real exploits

The first experiment ran the NullHttpd HTTP server. This server has a heap overflow vulnerability that an attacker can exploit by sending HTTP POST requests with a negative content length field [2]. These POST requests cause the server to allocate a heap buffer that is too small to hold the data in the request. While copying the data to the buffer, the server overwrites some data structures that are used by the C library to manage the heap. This vulnerability can be exploited to overwrite arbitrary words in memory.

We attacked NullHttpd using the technique described in [14]. The attack works by corrupting the CGI-BIN configuration string. This string identifies a directory holding programs that may be executed while processing HTTP requests. Therefore, by corrupting it, the attacker can force NullHttpd to run arbitrary programs. This is a non-control-data attack because the attacker does not subvert the intended control-flow in the server.

Data-flow integrity enforcement detects the attack on the first use of the CGI-BIN configuration string after it has been overwritten. It can do this because the library instructions that manipulate the internal heap data structures store invalid identifiers in the RDT entries for the words they write (as described in Section 2.4). This invalid identifier is, of course, not in the reaching definition set for the uses of the CGI-BIN configuration string.

In addition to preventing the attack, data-flow integrity enforcement found the following bug during the processing of NullHttpd's default configuration file:

If line contains the string "\n", the second pass through the loop accesses line [-1]. This could lead to memory corruption if line [-1] contained the character '\n' or '\r'.

Attack	Target data structure	Detected?
	parameter function pointer	yes
	parameter longjmp buffer	yes
direct overwrite on stack	return address	yes
direct overwrite on stack	old base pointer	yes
	function pointer	yes
	longjmp buffer	yes
direct overwrite on data segment	function pointer	yes
direct overwrite on data segment	longjmp buffer	yes
	parameter function pointer	yes
	parameter longjmp buffer	yes
overwrite through stack pointer	return address	yes
overwrite through stack pointer	old base pointer	yes
	function pointer	yes
	longjmp buffer	yes
	return address	yes
overwrite through data segment pointer	old base pointer	yes
overwine unough data segment pointer	function pointer	yes
	longjmp buffer	yes

Table 1: Synthetic attacks detected by data-flow integrity enforcement.

The second experiment ran SSH, which is a secure shell implementation from OpenSSH.org that contains an integer overflow vulnerability [3]. The problem is caused by an assignment of a 32-bit integer to a 16-bit integer. If the lower word of the 32-bit integer is 0, the 16-bit integer is assigned 0. This integer is then passed as an argument to malloc, which results in the allocation of a buffer that is too small. This vulnerability can be exploited to write any memory position in the server.

We ported the vulnerable SSH implementation to Windows and attacked it using the exploit in [14]. This exploit is similar to the example described in Section 2. It overwrites a variable, which is stored in the stack frame of another function, that records whether the user has been authenticated. The attacker can use the server without being authenticated by setting the variable to 1.

Data-flow integrity enforcement catches the attack on the first use of the authenticated variable after it has been overwritten, because the reaching definitions set for the use of the variable does not contain the identifiers of any definition outside the function where the variable is declared. This attack is detected even when data-flow integrity enforcement instruments only local variables without external definitions (like the version that we called *intraproc DFI* in the previous section).

The third experiment ran the GHTTP HTTP server, which has a number of vulnerabilities [1]. We ported GHTTP to Windows and exploited a buffer overflow vulnerability in a stack-based buffer to corrupt the return address. Data-flow integrity enforcement catches this common control-data attack because of the instrumentation that we add to check the integrity of return addresses (as described Section 2). The identifier stored by the buffer

write to the RDT entry of the return address is guaranteed to be different from the expected value of zero.

The final experiment ran Stunnel, which is a generic tunnel for security protocols. It is vulnerable to a format string attack [4] because it passes a string received from the network as a format string to the vsprintf function. To attack Stunnel, we supplied a format string that causes the return address of the function to be overwritten. This is typical of format string attacks and it is another control-data attack. As in the previous experiment, data-flow integrity enforcement detects the attack when the overwritten return address is about to be used by the return instruction.

5 Related work

Many mechanisms have been proposed to protect programs from attacks. CCured [30] and Cyclone [24] proposed safe dialects of C. They can prevent more attacks than data-flow integrity enforcement because they ensure memory safety for programs written in these dialects. However, they require a significant effort to port C applications to the safe dialects, and they require major changes to the runtime. For example, they replace malloc and free by a garbage collector. A garbage collector makes performance hard to predict and it can introduce a significant overhead, for example, the results presented in [30] show that CCured slows down the bc utility by almost a factor of 10.

Other mechanisms can only defend from attacks that overwrite specific targets, such as return addresses, function pointers or other control data (e.g., [18, 17, 35]). While effective against some attacks, these approaches

Application	Vulnerability	Exploit	Detected?
NullHttpd	heap-based buffer overflow	overwrite cgi-bin configuration data	yes
SSH	integer overflow and heap-based buffer overflow	overwrite authenticated variable	yes
STunnel	format string	overwrite return address	yes
Ghttpd	stack-based buffer overflow	overwrite return address	yes

Table 2: Real attacks detected by data-flow integrity enforcement.

can miss many control-data attacks [40] and they cannot defend against non-control data attacks.

Program Shepherding [27] and Control-Flow Integrity [5] are general mechanisms to ensure that a program does not deviate from its control-flow graph. They analyze the source code to compute a control-flow graph and use binary rewriting to enforce integrity of control-flow at runtime. Control-Flow Integrity provides lower overhead than Program Shepherding and it has been shown formally to catch any deviation from the control-flow graph [6]. However, as pointed out by [14], attacks can succeed without changing the control-flow of the target programs. Neither Control-Flow Integrity nor Program Shepherding can detect non-control-data attacks.

Several systems have proposed broad coverage attack detectors based on dynamic taint analysis [37, 15, 19, 31, 13, 16, 22, 33]. These mechanisms can detect both control-data and non-control-data attacks. They have the advantage of not requiring source code; they work on binaries and do not even require symbol information. However, they can have false positives and proposed implementations have high overhead or require hardware support.

There are several bounds checkers for C (e.g., [26, 36, 25, 34]). The Jones and Kelly [25] bounds checker does not require changes to the pointer format. It instruments pointer arithmetic to ensure that the result and original pointers point to the same object. To find the target object of a pointer, it uses a splay tree that keeps track of the base address and size of heap, stack, and global objects. A pointer can be dereferenced provided it points to a valid object in the splay tree. CRED [34] is similar but provides support for some common uses of out-ofbounds pointers in existing C programs. These bounds checkers can detect both control-data and non-control data attacks but they do not prevent all bounds violations. For example, they cannot prevent attacks that exploit format string vulnerabilities or that overwrite data using a pointer to a dead object whose memory was reused. Additionally, they have high overhead because of accesses to the splay tree. Data-flow integrity enforcement can prevent these types of attack and it has lower overhead.

Several systems use static analysis to detect software vulnerabilities in C and C++ programs (e.g., [38, 11, 12]). These systems have been very successful at finding vulnerabilities in existing software and they have the ad-

vantage of detecting problems at compilation time. Like data-flow integrity enforcement, they can miss attacks due to analysis imprecision but, unlike it, they can generate a large number of false positives. Other systems have proposed static techniques to enforce user-specified data confidentiality and integrity policies (e.g., [41]). Our goal is different: we want to enforce a low level safety property that is automatically derived from the source code of programs written in an unsafe language.

Debugging tools like Purify [23] can detect many memory errors in programs. They could be used to detect attacks but they have high overhead.

6 Conclusions

We introduced a simple safety property called data-flow integrity and presented an implementation that instruments programs automatically to enforce this property. Since most attacks must violate data-flow integrity to be successful, data-flow integrity enforcement can protect software from most attacks: it can prevent both control-data and non-control-data attacks. Previously, there was no good defense against non-control-data attacks and they will become common once we deploy defenses against control-data attacks.

Our implementation can be used in practice because it can be applied automatically to C and C++ programs without modifications and it does not have false positives. We described several optimizations that reduce the overhead of the instrumentation. These optimizations make it practical to run data-flow integrity enforcement in many applications.

We believe that it is possible to improve the coverage and performance of data-flow integrity enforcement. There has been recent work on more precise points-to analysis that can scale to large programs [12]. We could leverage this work to improve the precision of our interprocedural analysis.

Acknowledgments

We thank our shepherd Robbert van Renesse, Andrew Myers, Antony Rowstron, Paul Barham, Michael Fetterman, Jon Crowcroft and the anonymous reviewers for helpful comments on this work. We thank John Wilander for sharing his library of attacks with us.

References

- GHttpd Log() Function Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5960.
- [2] Null HTTPd Remote Heap Overflow Vulnerability. http://www.securityfocus.com/bid/5774.
- [3] SSH CRC-32 Compensation Attack Detector Vulnerability. http://www.securityfocus.com/bid/2347.
- [4] STunnel Client Negotiation Protocol Format String Vulnerability. http://www.securityfocus.com/bid/3748.
- [5] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow Integrity: Principles, implementations, and applications. In ACM CCS (Nov. 2005).
- [6] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. A theory of secure control flow. In *ICFEM* (July 2005).
- [7] AHO, A. V., SETHI, R., AND ULLMAN, J. D. Compilers: Principles, Techniques and Tools. Addison Wesley, 1986.
- [8] AMME, W., BRAUN, P., ZEHENDNER, E., AND THOMASSET, F. Data dependence analysis of assembly code. In *PACT* (Oct. 1998).
- [9] ANDERSEN, L. Program analysis and specialization for the C programming language. PhD thesis, University of Copenhagen, 1994.
- [10] APPEL, A. W. Modern Compiler Implementation in Java. Cambridge University Press, 1998.
- [11] ASHCRAFT, K., AND ENGLER, D. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *IEEE Sympo*sium on Security and Privacy (May 2002).
- [12] AVOTS, D., DALTON, M., LIVSHITS, V. B., AND LAM, M. S. Improving software security with a C pointer analysis. In *ICSE* (May 2005).
- [13] CHEN, S., XU, J., NAKKA, N., KALBARCZYK, Z., AND IYER, R. K. Defeating memory corruption attacks via pointer taintedness detection. In DSN (July 2005).
- [14] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *USENIX Security Symposium* (July 2005).
- [15] COSTA, M., CROWCROFT, J., CASTRO, M., AND ROWSTRON, A. Can we contain Internet worms? In *HotNets* (Nov. 2004).
- [16] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-toend containment of Internet worms. In SOSP (Oct. 2005).
- [17] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium* (Aug. 2003).
- [18] COWAN, C., PU, C., MAIER, D., HINTON, H., WADPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic detection and prevention of bufferoverrun attacks. In *USENIX Security Symposium* (Jan. 1998).
- [19] CRANDALL, J. R., AND CHONG, F. T. Minos: Control data attack prevention orthogonal to memory model. In MICRO-37 (Dec. 2004).
- [20] DEBRAY, S. K., MUTH, R., AND WEIPPERT, M. Alias analysis of executable code. In *POPL* (Jan. 1998).
- [21] HEINTZE, N., AND TARDIEU, O. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *PLDI* (June 2001).
- [22] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical taint-based protection using demand emulation. In *EuroSys* (Apr. 2006).

- [23] IBM. Purify. http://www-306.ibm.com/software/awdtools/ purify.
- [24] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CH-ENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In USENIX Annual Technical Conference (June 2002).
- [25] JONES, R., AND KELLY, P. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third Inter*national Workshop on Automated Debugging (May 1997).
- [26] KENDALL, S. Bcc: run-time checking for C programs. In USENIX Summer Conference (June 1983).
- [27] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In USENIX Security Symposium (Aug. 2002).
- [28] MCCORKENDALE, B., AND SZOR, P. Code Red Buffer Overflow. Virus Bulletin (Sept. 2001).
- [29] MICROSOFT. Phoenix compiler framework. http://research.microsoft.com/phoenix/phoenixrdk.aspx.
- [30] NECULA, G., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. CCured: Type-Safe Retrofitting of Legacy Software. ACM Transactions on Programming Languages and Systems 27, 3 (May 2005).
- [31] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In NDSS (Feb. 2005).
- [32] ONE, A. Smashing the stack for fun and profit. *Phrack* 7, 49 (Nov. 1996).
- [33] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys* (Apr. 2006).
- [34] RUWASE, O., AND LAM, M. A practical dynamic buffer overflow detector. In NDSS (Feb. 2004).
- [35] SMIRNOV, A., AND CHIUEH, T. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In NDSS (Feb. 2005).
- [36] STEFFEN, J. L. Adding run-time checking to the portable C compiler. Software Practice and Experience 22, 4 (Apr. 1992), 305–306
- [37] SUH, G. E., LEE, J., AND DEVADAS, S. Secure program execution via dynamic information flow tracking. In ASPLOS XI (Oct. 2004).
- [38] WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. A first step towards automated detection of buffer overrun vulnerabilities. In NDSS (Feb. 2000).
- [39] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In SOSP (Dec. 1993).
- [40] WILANDER, J., AND KAMKAR, M. A comparison of publicly available tools for dynamic buffer overflow prevention. In NDSS (Feb. 2003).
- [41] ZHENG, L., CHONG, S., MYERS, A. C., AND ZDANCEWIC, S. Using Replication and Partitioning to Build Secure Distributed Systems. In *IEEE Symposium on Security and Privacy* (May 2003).

From Uncertainty to Belief: Inferring the Specification Within

Ted Kremenek[†] Paul Twohey[†] Godmar Back[‡] Andrew Ng[†] Dawson Engler[†]

Computer Science Department[†] Stanford University Stanford, CA, U.S.A. Computer Science Department[‡] Virginia Tech Blacksburg, VA, U.S.A.

Abstract

Automatic tools for finding software errors require a set of specifications before they can check code: if they do not know what to check, they cannot find bugs. This paper presents a novel framework based on factor graphs for automatically inferring specifications directly from programs. The key strength of the approach is that it can incorporate many disparate sources of evidence, allowing us to squeeze significantly more information from our observations than previously published techniques.

We illustrate the strengths of our approach by applying it to the problem of inferring what functions in C programs allocate and release resources. We evaluated its effectiveness on five codebases: SDL, OpenSSH, GIMP, and the OS kernels for Linux and Mac OS X (XNU). For each codebase, starting with zero initially provided annotations, we observed an inferred annotation accuracy of 80-90%, with often near perfect accuracy for functions called as little as five times. Many of the inferred allocator and deallocator functions are functions for which we both lack the implementation and are rarely called — in some cases functions with at most one or two callsites. Finally, with the inferred annotations we quickly found both missing and incorrect properties in a specification used by a commercial static bug-finding tool.

1 Introduction

Many effective tools exist for finding software errors [4, 5,8,15,18,28,33]. While different in many respects, they are identical in one: if they do not know what to check, they cannot find bugs. In general, tools require specifications that document what a program *should do* in order for the tool to discern good program behavior from bad. Undetected errors due to missing specifications are a serious form of false negatives that plague sound and unsound bug-finding tools alike. From our own experience with developing such tools, we believe that legions of bugs remain undetected in systems previously "vetted" by checking tools simply because they lack the required specifications, and not because tools lack the necessary analysis precision.

Furthermore, checking tools generally operate without safety nets. There is no mechanism to discover when bugs in the checking tool lead to missed errors in checked code. Analysis bugs are a source of false negatives; while an unsound tool may have false negatives by design, even a sound tool can have false negatives due to implementation bugs. In our experience this is a serious concern: if an analysis bug does not cause a false positive, the only way to catch it is by comparison against regression runs.

The result of all these factors is that checking tools miss many bugs they could catch. Unfortunately, acquiring accurate specifications can be daunting at best. Even with state-of-the-art annotation systems, the manual labor needed to specify high-level invariant properties for large programs can be overwhelming [12, 35]. Further, in large, evolving codebases with many developers, interfaces may change rapidly as functions are added and removed. This churn exacerbates the problem of keeping a specification, if there is one, current.

Fortunately, there are many sources of knowledge, intuitions, and domain-specific observations that can be automatically leveraged to help infer specifications from programs directly. First and foremost, the behavior of programs is well-structured, with recognizable patterns of behavior implying high-level roles for the objects in a program. For example, functions that allocate and release resources (such as file handles and memory) vary widely in their implementations, but their interfaces are used nearly identically. In essence, the behavior of a program reflects what the programmer intended it to do, and the more we observe that one or more objects appear to interact in a recognizable role, the more we believe that role reflects their true purpose. More plainly, the more something behaves like an X the more we believe it is an X. Thus, leveraging such information has the desired property that the amount of evidence garnered about a program's specification grows in the amount of code analyzed. In addition, we often have a volume of valuable, non-numeric ad hoc information such as domain-specific naming conventions (e.g., a function name containing the word "alloc" often implies the function is an allocator).

This paper presents a novel, scalable, and customizable framework for automatically inferring specifications from programs. The specifications we infer come in the form of annotations, which are able to describe many kinds of important program properties. The key strength of our approach is that it tightly binds together many disparate sources of evidence. The result is that any information about one object becomes indirect information about related objects, allowing us to squeeze sig-

nificantly more information from our observations than previously published approaches. Our framework can incorporate many sources of information: analysis results from bug-finding tools, ad hoc knowledge, and already known annotations (when available). Moreover, because the technique is built upon a probabilistic model called a *factor graph* [21,36], it is fully capable of fusing multiple sources of information to infer specifications while handling the inherent uncertainty in our information sources and the often noisy relationships between the properties we wish to infer. Further, inferred annotations can be immediately employed to effectively find bugs even *before* the annotations are inspected by a user.

This last feature makes our framework pragmatic even for rapidly evolving codebases: the process of inferring annotations and using those annotations to check code with automated tools can be integrated into a nightly regression run. In the process of the daily inspection of bug reports generated by the nightly run, some of the inferred annotations will be inspected by a user. These now known annotations can subsequently be exploited on the next nightly regression to better infer the remaining uninspected annotations. Thus, our framework incrementally accrues knowledge about a project without a huge initial investment of user labor.

This paper makes the following contributions.

- We present Annotation Factor Graphs (AFGs), a group of probabilistic graphical models that we have architected specifically for inferring annotations.
- We illustrate how information from program analysis as well as different kinds of ad hoc knowledge can be readily incorporated into an AFG.
- We reduce the process of inferring annotations with an AFG to factor graph inference and describe important optimizations specific to incorporating information from static program analysis into an AFG.
- 4. We provide a thorough evaluation of the technique using the example of inferring resource management functions, and illustrate how our technique scales to large codebases with high accuracy. Further, we show that with our results we have found both missing and incorrect properties in a specification used by a commercial bug-finding tool (Coverity Prevent [7]).

Section 2 presents a complete example of inferring specifications from a small code fragment, which introduces the key ideas of our approach. Section 3 uses the example to lay the theoretical foundations, which Section 4 further formalizes. Section 5 refines the approach and shows more advanced AFG modeling techniques. Section 6 discusses the computational mechanics of our inference technique. We evaluate the effectiveness of our approach in Section 7, discuss related work in Section 8, and then conclude.

```
FILE * fp = fopen("myfile.txt","r");
fread( buffer, n, 1000, fp );
fclose( fp );
```

Figure 1: Simple example involving a C file handle: fopen "allocates" the handle, fread uses it, and folose releases it.

2 Motivating Example

We now present a complete example of inferring specifications from a small code fragment. This example, along with the solution presented in the next section, serves to introduce many of the core ideas of our technique.

2.1 Problem: Inferring Ownership Roles

Application and systems code manages a myriad of resources such as allocated heap objects, file handles, and database connections. Mismanagement of resources can lead to bugs such as resource leaks and use-after-release errors. Although tools exist to find such bugs [17, 18, 28, 32], all require a list of functions that can allocate and release resources (allocators and deallocators). Unfortunately, systems code often employs non-standard APIs to manage resources, causing tools to miss bugs.

A more general concept that subsumes knowing allocation and deallocation functions is knowing what functions return or claim ownership of a resource. Many C programs use the ownership idiom: a resource has at any time exactly one *owning* pointer, which must release the resource. Ownership can be transferred from a pointer by storing it into a data structure or by passing it to a function that claims it. A function that returns an owning pointer has the annotation *ro* (*returns ownership*) associated with its interface. A function that claims a pointer passed as an argument has the property *co* (*claims ownership*) associated with the corresponding formal parameter. In this model, allocators are *ro* functions, while deallocators are *co* functions.

Many *ro* functions have a contract similar to an allocator, but do not directly allocate resources. For example, a function that dequeues an object from a linked list and returns it to the caller. Once the object is removed from the list, the caller must ensure the object is fully processed. A similar narrative applies to *co* functions.

Consider the task of inferring what functions in a program return and claim ownership of resources. For example, assume we are given the code fragment in Figure 1 and are asked to determine if fopen is an ro and if either fread or fclose are co's. Without prior knowledge about these functions, what can we conclude by looking at this fragment alone?

While we cannot reach a definitive conclusion, simple intuition renders some possibilities more likely than others. Because programs generally behave correctly, a person might first make the assumption that the code fragment is *likely* to be bug-free. This assumption elevates

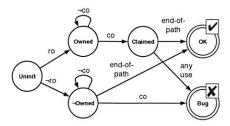


Figure 2: DFA summarizing basic ownership rules. A pointer returned from a function call enters either the Owned or $\neg Owned$ state depending on whether the called function has the property ro or $\neg ro$ respectively. Function calls involving the pointer cause the DFA to transition states based on the co or $\neg co$ property associated with the called function. An "end-of-path" indicates no further uses of the pointer within the function. The two final states for the DFA indicate correct (OK) or incorrect use of the pointer (Bug).

the likelihood of two conclusions over all others.

First, fopen may be an ro, fread a function that uses fp but does not claim ownership of the resource ($\neg co$), and fclose a co. For this case, the code fragment can be logically rewritten as:

$$fp = ro(); \neg co(fp); co(fp)$$

This conclusion follows if we assume the code fragment is correct. If fopen is an ro, then this assignment is the only one to all three functions that does not induce a bug. To avoid a resource-leak, either fread or fclose must be a co. To avoid a use-after-release error, however, fread cannot be a co. This leaves fclose being a co.

Here we have assumed that an owned pointer cannot be used after being claimed. This strict interpretation of the ownership idiom assumes that all co's are deallocators. For now, the correctness rules for ownership that we use are summarized by the DFA in Figure 2, and we discuss refinements in Section 5.1

Continuing, the second likely assignment of ownership roles to these functions is that fopen is an $\neg ro$, with both fread and fclose as $\neg co$'s:

$$fp = \neg ro(); \neg co(fp); \neg co(fp);$$

We reach this conclusion using similar reasoning as before. If fopen is an $\neg ro$, we no longer have the opportunity to leak an allocated resource, but as Figure 2 shows, it is now a bug to pass fp to a co.

Consequently, from simple reasoning we can infer much about these functions. Note that we are not *certain* of anything; we only *believe* that some conclusions are more likely than others. Further, we may be able to infer more about these functions by incorporating additional intuitions and knowledge. In the next section we discuss how to precisely formulate such reasoning.

3 The Big Picture

This section gives a crash course in our inference approach, tying it to the example just presented. By its end,

the reader will have a basic arsenal for inferring annotations, which the next sections extend.

Our goal is to provide a framework (a probabilistic model) that (1) allows users to easily express every intuition and domain-specific observation they have that is useful for inferring annotations and then (2) reduces such knowledge in a sound way to meaningful probabilities. In the process, the framework squeezes out all available information about the annotations we are inferring.

The inference framework must solve two common challenges. First, it must robustly handle noise. Otherwise its brittleness will prevent the exploitation of many observations that are only "often" true rather than always true (e.g., an observation that an annotation obeys a feature 51% of the time). Second, it must soundly combine uncertain information. Treated independently, a fact of which we are only partially certain is only marginally useful. However, aggregated with other sources of knowledge, one fact can be the missing link in a chain of evidence we would not otherwise be able to exploit.

Our annotation inference has three steps:

- 1. **Define** the set of possible annotations to infer.
- 2. **Model** domain-specific knowledge and intuitions in our probabilistic model.
- Compute annotation probabilities using the model.
 These probabilities are then used to rank, from most-to-least probable: entire specifications, individual annotations, or errors.

We now apply these three steps to our motivating example. In general, annotations are implicitly defined by a correctness property that we wish to check in a program; we briefly describe it for our example (\S 3.1). We then describe the fundamental mechanics of our probabilistic model (\S 3.2), and how to model two basic properties: (1) the fact that the more something behaves like an X the more likely it is an X (\S 3.3) and (2) domain-specific prior beliefs (\S 3.4). We finish the section by computing the probabilities for the annotations in our example.

3.1 Defining the Annotations to Infer

We use annotation variables to denote the program objects to which annotations bind. The example has three such variables: fopen:ret, fread:4 and fclose:1. The variable fopen:ret corresponds to the possible annotations for the return value of fopen, and has the domain $\{ro, \neg ro\}$. The variables fread:4 and fclose:1 (where ":i" denotes the ith formal parameter) have the domain $\{co, \neg co\}$. We have $2^3 = 8$ combinations of values for these variables, and each represents a joint specification of the roles of all three functions. (For this paper, an annotation and its negation are mutually exclusive.) We denote the set of annotation variables as A. In our example, $A = \{fopen:ret, fread:4, fclose:1\}$. Further, the notation

ANNOTATIONS			AN	IS	FACTOR VALUES				FACTOR VALUES				
fopen:ret fread	fread:4	f:4 fclose:1	DFA	$f_{\langle check \rangle}$	$f_{\langle ro \rangle}(fopen:ret)$	$f_{\langle co \rangle}(fread:4)$	$f_{\langle co \rangle}(fclose:1)$	$\prod f$	$\frac{1}{Z}\prod f$				
ro	$\neg co$	co	V	0.9	0.8	0.7	0.3	0.151	0.483				
$\neg ro$	$\neg co$	$\neg co$	V	0.9	0.2	0.7	0.7	0.088	0.282				
ro	$\neg co$	$\neg co$	×	0.1	0.8	0.7	0.7	0.039	0.125				
ro	co	$\neg co$	×	0.1	0.8	0.3	0.7	0.017	0.054				
ro	co	co	×	0.1	0.8	0.3	0.3	0.007	0.023				
$\neg ro$	$\neg co$	co	×	0.1	0.2	0.7	0.3	0.004	0.013				
$\neg ro$	co	$\neg co$	×	0.1	0.2	0.3	0.7	0.004	0.013				
$\neg ro$	co	co	×	0.1	0.2	0.3	0.3	0.002	0.006				

Table 1: Table depicting intermediate values used to compute P(fopen:ret, fread:4, fclose:1). Specifications are sorted by their probabilities. The value Z is a normalizing constant computed by summing the column $\prod f$. By summing the values in the last column when fopen:ret = ro we compute $P(fopen:ret = ro) \approx 0.7$. The marginal probabilities $P(fread:4 = \neg co) \approx 0.9$ and $P(fclose:1 = co) \approx 0.52$ are similarly computed.

A = a means that a is some concrete assignment of values to *all* the variables in A. Thus a represents one possible complete specification, which in our example specifies ownership roles for all three functions.

Table 1 lists all possible specifications for Figure 1, and displays the intermediate values used to compute probabilities for each specification. We use this table throughout the remainder of this section. The "DFA" column indicates, for each specification, the final state of the DFA from Figure 2 when applied to the code example.

3.2 Modeling Beliefs for Annotations

The rest of the section shows how to express domaininsights in our inference framework in terms of user-defined mathematical functions, called *factors*. The framework uses these factors to compute annotation probabilities, e.g. the probability P(fopen:ret = ro) that the return value of fopen has the annotation ro.

Factors are relations mapping the possible values of one or more annotation variables to non-negative real numbers. More precisely, each factor f_i is a map from the possible values of a set of variables \mathbf{A}_i ($\mathbf{A}_i \subseteq \mathbf{A}$) to $[0,\infty)$. Factors "score" an assignment of values to a group of related annotation variables, with higher values indicating more belief in an assignment. Although not required, for many factors the mapped values we specify are probabilities (i.e. sum to 1) because probabilities are intuitive to specify by hand.

Suppose we know that functions in a codebase that return a value of type <code>FILE*</code> have a higher chance of returning ownership (ro). We can express this tendency using a factor that, given the annotation label for the return value of a function with a <code>FILE*</code> return type, returns slightly higher weights for ro annotations than $\neg ro$ annotations. For example, making $f_{\texttt{FILE}*}$ (fopen:ret = ro) = 0.51 and $f_{\texttt{FILE}*}$ (fopen:ret $= \neg ro$) = 0.49. Note that the magnitude of the values is not important, but rather their relative odds. We could have used 51 and 49 instead, implying an odds of 51:49 to prefer ro over $\neg ro$ annotations. While trivial, this example illustrates a couple of recurrent themes. First, the observations we want to exploit for inference are often tendencies and not laws — as long as these tendencies are right more often than wrong,

they convey useful information. The ratio that their associated factor returns reflects the reliability of a tendency—big ratios for very reliable tendencies, smaller ratios for mildly reliable ones. Second, factors work well in our domain since (1) they let the user express any computable intuition in terms of a custom function and (2) inference intuitions naturally reduce to preferences over the possible values for different groups of annotation variables. Factors thus provide a way to reduce disparate observations to a common currency.

Once we represent individual beliefs with factors, we combine a group of factors $\{f_i\}$ into a single probability model by multiplying their values together and normalizing:

$$P(\mathbf{A}) = \frac{1}{Z} \prod_{f_i \in \{f_j\}} f_i(\mathbf{A}_i) \tag{1}$$

The normalizing constant Z causes the scores to define a probability distribution over the values of A, which directly translates into a distribution over specifications.

One way to view Equation 1 is as a specific case of Hinton's concept of a Product of Experts [19]. From this view, each factor represents an "expert" with some specific knowledge which serves to constrain different dimensions in a high-dimensional space. The product of these experts will then constrain all of the dimensions. In this case, the dimensions represent the space of possible specifications, and the most probable specifications are those that most "satisfy" the combined set of constraints.

Note that because factors can share inputs their values are not independent; such correlation gives the model much expressive power. As stated previously (§ 2.1), annotations may be correlated (e.g., if fopen is an ro, then fclose is likely a co). We capture this correlation by having the probabilistic model represent a probability distribution P(A) over the values of all the annotation variables A.

The simplicity of Equation 1 is deceptive. It is trivial to define an ad hoc scoring function that in a deep sense means nothing. This function is not one of those. It subsumes the expressive power of both Bayesian networks and Markov Random Fields, two widely used methods for modeling complex probabilistic domains, because

both can be directly mapped to it. In our experience it is also simpler and easier to reason about. We defer discussion of its mathematical foundation (see [36]) — but for our purposes, it has the charm of being understandable, general enough to express a variety of inference tricks, and powerful enough to make them mean something.

We now show how to use factors by defining two common types of factors for our example: behavioral signatures and prior beliefs.

3.3 Behavioral Signatures

"Behavior is the mirror in which everyone shows their image."

— Johann Wolfgang von Goethe

In general, the more something behaves like an X, the more probable it is an X. In our context, programmers often use program objects that should have a given annotation in idiomatic ways. Such *behavioral signatures* provide a good source of data that we can automatically extract from code (e.g., by using static analysis), which gives a nice mechanical way to get large data samples. As with any statistical analysis, the more data we have the more information we can extract.

The most common behavioral signature for any annotation is that programs generally behave correctly and that errors are rare. Thus, correct annotations (typically!) produce fewer errors than incorrect ones. We measure how much a behavioral signature is exhibited in a program by using a *behavioral test*. A behavioral test works as follows: for every location in a program where a behavioral signature *may* apply, we conduct a *check* to see if a given set of annotations matches that signature. Because the behavior may involve reasoning about the semantics of different execution paths in the program (either intra- or inter-procedurally), this check is implemented using a static analysis *checker*.

The checker we use in this paper is an intra-procedural static analysis that simulates the DFA in Figure 2 on paths through functions stemming from callsites where a pointer is returned. In principle, however, checks can employ program analysis of arbitrary complexity to implement a behavioral test, or even be done with dynamic analysis. The output of the checker indicates whether or not the annotation assignment $\mathbf{a_i}$ to the set of annotation variables $\mathbf{A_i}$ matched with one (or more) behavioral signatures. We can capture this belief for our example with a single *check factor*, $f_{\langle check \rangle}$ (fopen:ret, fread:4, fclose:1):

$$f_{\langle check \rangle}(...) = \left\{ \begin{array}{ll} \theta_{\langle ok \rangle} & : & \text{if DFA} = \text{OK} \\ \theta_{\langle bug \rangle} & : & \text{if DFA} = \text{Bug} \end{array} \right.$$

This factor weighs assignments to *fopen:ret*, *fread:4*, and *fclose:1* that induce bugs against those that do not with a ratio of $\theta_{\langle \text{bug} \rangle}$ to $\theta_{\langle \text{ok} \rangle}$. For instance, suppose we believe that any random location in a program will have a

```
    FILE * fpl = fopen( "myfile.txt", "r" );
    FILE * fp2 = fdopen( fd, "w" );
    fread( buffer, n, 1, fpl );
    fwrite( buffer, n, 1, fp2 );
    fclose( fp1 );
    fclose( fp2 );
```

Figure 3: Code example that would produce two distinct checks, one for fpl and fp2 respectively. Observe that both fopen and fdopen return ownership of a resource that must be claimed by fclose.

bug 10% of the time. This can be reflected in $f_{\langle check \rangle}$ by setting $\theta_{\langle \text{bug} \rangle} = 0.1$ and $\theta_{\langle \text{ok} \rangle} = 0.9$. These values need only be rough guesses to reflect that we prefer annotations that imply few bugs.

Check factors easily correlate annotation variables. In this case, $f_{\langle check \rangle}$ correlates fopen:ret, fread:4, and fclose:1 since its value depends on them. In general there will be one " $f_{\langle check \rangle}$ " for each location in the program where a distinct bug could occur. Figure 3 gives an example with two callsites where two file handles are acquired by calling fopen and fdopen respectively. These two cases constitute separate checks (represented by two factors) based on the reasoning that the programmer made a distinct decision in both cases in how the returned handles were used. Observe in this case that the variable fclose:1 is associated with two checks, and consequently serves as input to two check factors. Thus the more a function is used the more evidence we acquire about the function's behavior through additional checks.

3.4 Prior Beliefs: Small Sample Inference

As in any data analysis task, the less data we have the harder inference becomes. In our domain, while a large number of callsites are due to a few functions, a large number of functions have few callsites. As already discussed, AFGs partially counter this problem by explicitly relating different functions together, thus information from one annotation can flow and influence others. An additional approach that cleanly melds into our framework is the use of *priors* to indicate initial preferences (biases). We attach one prior factor to each annotation variable to bias its value. Having one prior factor per annotation variable, rather than per callsite, means it provides a constant amount of influence that can be gradually overwhelmed as we acquire more evidence.

For example, a completely useless specification for a codebase is that all functions have either $\neg ro$ or $\neg co$ annotations. This specification generates no bugs since nothing is ever allocated or released, but is vacuous because we are interested in identifying allocators and deal-locators. One counter to this problem (we discuss an additional method later in § 5.1) is to encode a slight bias towards specifications with ro's and co's. We encode this bias with two sets of factors. The first is to bias towards

ro annotations with the factor $f_{\langle ro \rangle}$:

$$f_{\langle ro \rangle}(X) = \left\{ \begin{array}{ll} \theta_{\langle ro \rangle} & : & \text{if } X = ro \\ \theta_{\langle \neg ro \rangle} & : & \text{if } X = \neg ro \end{array} \right.$$

For instance, $\theta_{\langle ro \rangle} = 0.8$ and $\theta_{\langle \neg ro \rangle} = 0.2$ means that we prefer ro to $\neg ro$ annotations with an odds of 8:2. Our example has one such factor attached to *fopen:ret*. Values of this factor for different specifications are depicted in Table 1.

The addition of $f_{\langle ro \rangle}(fopen:ret)$ biases us towards ro annotations, but may cause us to be overly aggressive in annotating some formal parameters as co in order to minimize the number of bugs flagged by the checker (thereby maximizing the value of the $f_{\langle check \rangle}$ factors). To bias against co annotations, we define $f_{\langle co \rangle}$ in an analogous manner to $f_{\langle ro \rangle}$. We set $\theta_{\langle co \rangle} = 0.3$ and $\theta_{\langle \neg co \rangle} = 0.7$, which makes our bias away from co annotations slightly weaker than our bias for ro annotations. In our example, we add two co factors: $f_{\langle co \rangle}(fread:4)$ and $f_{\langle co \rangle}(fclose:1)$. These two factors, while distinct, share the same values of $\theta_{\langle co \rangle}$ and $\theta_{\langle \neg co \rangle}$.

In general, we use priors when we have some initial insights that can be overcome given enough data. Further, when we do not know enough to specify their values, they can be omitted entirely.

3.5 Computing Probabilities

Equipped with four factors to encode beliefs for the three functions, we now employ Equation 1 to compute final probabilities for the possible specifications.

We first multiply the values of the factors (shown in column " $\prod f$ " of Table 1) and then normalize those values to obtain our final probabilities. From Table 1, we can see that for the specification $\langle ro, \neg co, co \rangle$ (first row) the product of its individual factor values is $0.9 \times 0.8 \times 0.7 \times 0.3 = 0.151$. After normalizing, we have a probability of 0.483, making it the most probable specification. Observe that because of our bias towards ro annotations, the second most probable specification, $\langle \neg ro, \neg co, \neg co \rangle$, has a probability of 0.282 (almost half as likely). Further, we can use the table to compute the probabilities of individual annotations. To compute P(topen:ret = ro), we sum the probabilities in the last column for each row where topen:ret = ro.

These probabilities match our intuitive reasoning. The probability $P(fopen:ret=ro)\approx 0.7$ indicates that we are confident, but not completely certain, that fopen could be an ro, while $P(fread:4=\neg co)\approx 0.9$ shows that we have a strong belief that fread is $\neg co$. For fclose, we are left with a modest probability that $P(fclose:1=co)\approx 0.52$. While this number seems low, it incorporates both our bias towards ro annotations and our bias against co annotations. Observe that if we rank the possible co's by their probabilities (in this case only two), fclose is at

the top. The more locations where fclose is used in a similar manner the more confidently we believe it is a *co*.

3.6 How to Handle Magic Numbers?

By now, the reader may feel uneasy. Despite dressing them in formalisms, factors simply map annotation values to magic numbers. (Or, in formal newspeak, the *parameters* of the model.) As with all uses of magic numbers two questions immediately surface: (1) where do they come from? and (2) how hard are they to get right?

For our experiments, even crudely-chosen parameter values (as we have done so far) work well. This robustness seems to be due to two features: (1) the large amount of data that inference can work with and (2) the strong behavioral patterns evident in code. Further, these numbers seem portable across codebases. In a separate paper [20], when we took the parameters learned on one codebase A and used them for inference on another codebase B the results were imperceptible compared to learning them directly on B from a perfect set of known annotations. (We discuss the mechanics of learning in § 6.3.)

However, assume a hypothetical case where neither approach works. Fortunately, the way annotations get consumed provides algorithmic ways to converge to good parameter values. First, we commonly run inference repeatedly over a codebase as it evolves over time — thus, there will be significant numbers of known annotations from previous runs. Because our approach easily incorporates known information (in this case previously validated annotations), users can (1) apply machine learning to refine parameter values, (2) recompute annotation probabilities, or (3) do both simultaneously. In all cases results will improve.

The same approach works even when applying inference to a codebase for the first time. Annotations either get consumed directly or get fed to an error checking tool. In both cases we can sort the annotations or errors (based on the probabilities of their underlying annotations) from most-to-least probable, and inspect them by going down the list until the number of inference mistakes becomes annoying. We then rerun inference using the validated annotations as discussed above.

Second, it is generally clear how to conservatively bias parameters towards higher precision at the expense of lower recall. For example, assume we feed inferred annotations to a checker that flags storage leaks (ro without a subsequent co). We can strongly bias against the ro annotation to the extent that only a function with a very low error rate can get classified as an ro. While this misses errors initially, it ensures that the first round of inspections has high quality errors. Rerunning learning or inference will then yield a good second round, etc.

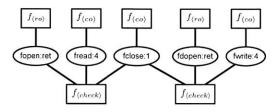


Figure 4: Factor graph for the code example in Figure 3. Rectangular nodes represent factors and round node represent annotation variables. The top factors represent factors for prior beliefs (§ 3.4), while the bottom factors represent behavioral tests (checks) (§ 5.1).

4 Annotation Factor Graphs

While Equation 1 is the formal backbone of our framework, in practice we operate at a higher level with its corresponding visual representation called a *factor graph*. A factor graph is an undirected, bipartite graph where one set of nodes represent the variables $\bf A$ and the other set of nodes represent the factors $\{f_i\}$.

We call the factor graphs we construct for annotation inference Annotation Factor Graphs (AFGs). Figure 4 depicts an AFG for the code example in Figure 3. Depicted are nodes for annotation variables, factors for prior beliefs (§ 3.4), and factors that represent two distinct checks of the code fragment. Each factor node (square) maps to a distinct factor multiplicand in Equation 1, each variable node (oval) maps to an annotation variable, and an edge exists between a factor node and variable node if the variable is an input to the given factor. Further, beliefs about one variable can influence another if there is a path between them in the graph. For example, observe that the AFG in Figure 4 explicitly illustrates the indirect correlation between fopen and fdopen through their relationship to fclose. If we believe that fopen is an ro, this belief propagates to the belief that fclose is a co because of the check for fp1. This belief then propagates to the belief that fdopen is an ro because of the check for fp2. While the AFG illustrates this flow of correlation, the underlying machinery is still Equation 1. Thus AFGs meet our criteria for an inference framework capable of combining, propagating, and reasoning about imperfect information.

A benefit of factor graphs is that they more compactly represent P(A) than the table in Section 3, which scales exponentially in the number of possible annotations. The inference algorithm we use to infer probabilities for annotations (§ 6.2) operates directly on the AFG, forgoing the need to build a table and exploiting the graphical structure to determine which features influence others.

5 Advanced Inference Techniques

Building on our concept of AFGs, this section goes beyond the basic inference techniques of Section 3 to more advanced themes. We keep our discussion concrete by exploring how to build factors for the ownership problem that incorporate multiple, differently-weighted behavioral tests (§ 5.1), exploit ad hoc naming conventions (§ 5.2), and handle a function that may have no good labeling for a specification because it grossly violates the ownership idiom (§ 5.3). Our experiments (§ 7) evaluate an AFG with all of these features. While we discuss these points in terms of the ownership problem, they readily apply to other inference tasks.

5.1 Multiple Behavioral Tests

The properties we wish to infer almost always have multiple behavioral signatures. For signatures that are mutually exclusive (i.e., only one behavior can be exhibited at once) we can define a single check factor that weighs the different observed behaviors. For non-exclusive behaviors, we can simply define different factors as we have done so far. We thus focus on the former. We illustrate the technique by refining the checker in Figure 2 from a checker which reduced all behaviors to two mutually exclusive states ($\{Bug, OK\}$) to one which captures more nuanced behavior with five final states, given in Figure 5. The five signatures it accepts are:

- 1. *Deallocator*: a ro's returned pointer reaches a single matching co that occurs at the end of the trace.
- Ownership: a ro's returned pointer reaches a single matching co that does not occur at the end of the trace (i.e., is followed by one or more ¬co functions).
- 3. Contra-Ownership: a $\neg ro$'s returned pointer only reaches $\neg cos$.
- Leak: an error where a ro's returned pointer does not reach a co.
- 5. *Invalid use*: an error, includes all misuses besides *Leak*.

We assign weights to these outcomes as follows. First, we bias away from the promiscuous assignment of ¬ro¬co (accepted by Contra-Ownership) and towards ro-co (accepted by Deallocator) by giving Deallocator twice the weight of Contra-Ownership. Since Deallocator is the harshest test, we use it as a baseline, giving it the weight 1.0 and, thus, Contra-Ownership the weight 0.5. The other non-error case, Ownership, is the least intuitive (and we arrived at it after some experimentation): we weigh it slightly less than Contra-Ownership since otherwise the AFG over-biases towards ro-co annotations.

As before, we weight error states less than non-error states because of the assumption that programs generally behave correctly (§ 2.1). Thus, for errors we assign a low score: 0.1 for leaks and 0.01 for all other bugs (the latter occurring very rarely in practice). (Note that even if a codebase had *no* errors we would still assign these outcomes non-zero values since static analysis imprecision can cause the checker to falsely flag "errors.")

Although all of these weights were specified with simple heuristics, we show in Section 7 that they provide respectable results. Moreover, such weights are amenable

EXTENDED-DFA STATE	$f_{\langle check \rangle}$	REASON FOR WEIGHT			
Deallocator	1.0	Baseline			
Contra-Ownership	0.5	Weight half as much as Deallocator			
Ownership	0.3	Slightly less than Contra-Ownership			
Leak	0.1	Low weight: errors are rare			
Invalid Use	0.01	Such errors are very rare			

Table 2: Values of $f_{\langle check \rangle}$ for the behavioral signatures for the ownership idiom. Observe that for this factor the values do not sum to 1, but instead are based reasoning about relative "reward."

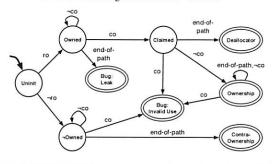


Figure 5: Complete DFA used by a static checker to implement behavioral tests for Ownership. Shaded nodes are error states.

to automatic tuning (§ 6.3).

5.2 Ad Hoc Knowledge: Naming Conventions

Ad hoc knowledge captures information such as biases for certain annotations, naming conventions, and other heuristics. We further illustrate how to express such features by using factors to model naming conventions.

Given a list of keywords suggestive of an ro annotation (e.g., "alloc", "open"), we construct a factor $f_{\langle keyword,ro \rangle}$ for each function whose name contains the keyword as a substring:

$$f_{\langle keyword, ro \rangle}(A) = \left\{ \begin{array}{ll} \theta_{\langle keyword, ro \rangle} & : & A = ro \\ \theta_{\langle keyword, \neg ro \rangle} & : & A = \neg ro \end{array} \right.$$

This factor is attached to a function's $ro/\neg ro$ annotation variable and rewards or penalizes ro annotations depending on the ratio of the factor's values. This factor (and its parameters) is replicated for all functions whose name contains the given keyword.

For co variables, the construction is similar, except that instead of the factor being associated with one ro annotation variable, it is associated with all the co annotation variables for a function. If any of those variables have the value co, then the factor maps to $\theta_{\langle keyword, \neg co \rangle}$ and to $\theta_{\langle keyword, \neg co \rangle}$ otherwise. Our reasoning is that naming conventions imply that at least one of the parameters should have a co annotation.

We only construct such factors if the function contains the keyword; in our experience while the presence of a keyword may suggest a function's role, the absence of a keyword usually provides no additional information.

5.3 Does the Model Fit?

The ownership idiom only approximates how developers use resources. Thus, there will be functions that do not fit

it (e.g., byzantine uses of reference counting). If a function is far enough outside the ownership model, then no matter what annotation(s) we assign it, ro or $\neg ro$, co or $\neg co$, the checker will flag a proliferation of "errors." This problem arises in many inference contexts: the property we want to infer assumes some structure of how code behaves, and when those assumptions are wrong, the resultant noise can hurt inference dramatically. AFGs let us tackle this problem by simply modeling the possibility that a function may not fit the ownership. We discuss two ways to do so.

The first way just adds another annotation $\neg fit$ to the domain of our annotation variables and models "not fitting" as yet another behavioral signature (as in § 5.1). We modify the checker described in Section 5.1 so that if it encounters a ¬fit annotation while checking it transitions to a special end-state Outside Model. For the ownership idiom, this increases the number of end-states in the checker DFA from five to six. We then change $f_{(check)}$ to assign Outside Model a value slightly above those for errors but below that of acceptable states (e.g., 0.2). Thus, if all values for an annotation variable cause many errors, we will overcome our bias against the $\neg fit$ value and classify the variable as not fitting the model. Once an annotation variable is classified as $\neg fit$, all behavioral tests involving that function are effectively discarded since the outcomes of those checks will always be Outside Model. Thus, Outside Model represents a state where the checker accepts any behavior, regardless of the annotation values.

The second way treats the not-fit property more globally: if the annotation variables for a function's return value or any of its formal parameters cause it to not fit, we remove the entire function from our checks. Implementing this requires correlating annotation variables in the AFG in a new way by adding a "meta-annotation" for each function whose corresponding annotation variable has the domain $\{fit, \neg fit\}$. For example, for the function fopen we would create the variable fopen:fit. Similarly as above, we modify the checker to include an Outside Model state, but because the not-fit property is now shared by all of the function's annotation variables, the checker consults the fit-variable for a function immediately prior to consulting the values of any of the function's regular annotation variables. As before, when the value is $\neg fit$ the checker transitions to *Outside Model*; otherwise execution continues as normal by consulting the regular annotation variable and taking the usual transition in the DFA. Consequently, a function's fit-variable serves as an additional input to each check factor that involves at least one of the function's regular annotation variables. As before, we state a strong bias toward a value of fit by attaching a prior belief factor to the fitvariable. Thus, if all values for any of a function's annotation variables lead to many errors, we overcome our bias against the $\neg fit$ value and classify the function as not fitting the model.

As described in Section 7.3, this second construction found aspects of Linux that proved noisome for inference. Moreover, neither approach is specific to the ownership idiom and can be applied to other annotation inference domains.

6 Implementation

We now discuss our implementation, including the details of the actual checker and how probabilities are computed in practice from the AFG.

6.1 The Checker

Our program analysis is built on CIL [27] and is similar to xgcc [15]: unsound, but efficient and reasonably effective. The analysis engine, written in OCaml, is partially derived from the version of xgcc as described in Chou's thesis [6]. Our entire factor graph implementation is written in OCaml as well.

For the ownership checker, beyond constructing checks for every callsite that returns a pointer, we track the use of string constants within the function and treat them as if they were returned by a $\neg ro$. Pointer dereferences are modeled as implicit calls to a ¬co function to monitor if an owning pointer is used in any way after it is claimed. We also perform minor alias tracking: when tracking a pointer p and we observe the expression g = p (where g is a local variable) we add g to p's alias set. We do not model ownership transfer from p to q (unlike [18]) and simply treat q as another name for p since both pointers are local variables (i.e., the reference has not escaped the function). Further, when a tracked pointer is involved in a return statement, we consult the corresponding ro ($\neg ro$) annotation of the enclosing function and treat it essentially as a $co(\neg co)$ annotation except that for error conditions we always transition to the Invalid Use state. This follows from the intuition that while programmers may accidentally leak resources on unforeseen paths, when implementing a function they reason very consciously about the properties of the function's return value. Hence ownership errors at return sites are considered very rare. This allows the AFG to model a form of inter-procedural reasoning that handles idioms such as wrapper functions.

Because the checker analyzes multiple paths stemming from a single pointer returning callsite, we summarize the results over all analyzed paths by reporting the most pessimistic DFA state found: *Invalid Use*, followed by *Leak*, *Ownership*, *Contra-Ownership*, and finally *Deallocator*. The idea is to penalize annotations that induce errors, and reward annotations that completely obey our strictest behavioral signatures.

```
\triangleright A: annotation variables, known: known annotations
GIBBSSAMPLE(A, known)
  a = \{\}

    initial random values

   for v \in A, v \not\in known
       a[v] = RANDOMVALUE(DOMAIN(v))
  burn in the sample
  for j = 1 to 1000
       for v \in PERMUTEVARIABLES(A), v \notin known
           N = 0.0
           \triangleright compute all scores that rely on v's annotation
           for d \in Domain(v)
                scores = \{\}
                a[v] = d
                scores[d] = FactorsScore(v, a)
                N = N + score[d]
           for d \in Domain(v)
                scores[d] = scores[d]/N
           a[v] = DISTRIBUTIONSAMPLE(scores)
  return a
```

Figure 6: Pseudocode for Gibbs sampling from P(A).

Finally, there are paths the checker does not analyze because they are beyond its limited reasoning capability. We abort the analysis of paths where a tracked pointer is stored either into a global or the field of a structure. If all the paths for a given check would be discarded (this is determined when the AFG is constructed) the check is not included in the AFG. While this leads to some selection bias in the paths we observe, our decision was to focus on paths that would provide the cleanest evidence of different behavioral signatures. Accurately modeling the heap shape of systems programs statically is an important open problem that we (fortunately) did not need to solve in order to infer properties of many functions.

6.2 Probabilistic Inference

In theory it is possible to compute probabilities for annotations by directly using Equation 1, but this requires enumerating an exponential number of combinations for the values of annotation variables. Instead, we estimate probabilities using *Gibbs sampling* [13], which generates approximate samples from the distribution P(A). Abstractly this algorithm simulates a random walk through a Markov chain. Theory shows that once the chain is run long enough it converges to its stationary distribution (equilibrium), and sampling from that distribution is equivalent to drawing samples from P(A).

While the full details of Gibbs sampling are beyond the scope of this paper, the pseudocode for generating samples is depicted in Figure 6. Gibbs sampling has two key advantages over other algorithms: (1) it treats the checker implementation as a black box and (2) at all times there is a complete assignment of values (in *a*) to all the variables in the AFG. The upshot is that we always run the checker with a full set of annotations.

To generate a single sample, we perform "burn-in," a

process of iteratively adjusting the values of the variables to drift them towards a configuration consistent with being drawn from P(A). Determining the *minimum* number of iterations for the burn-in is generally perceived as a black art; we have chosen a large number (1000) that has yielded consistently reliable results.

Each iteration of the burn-in visits each variable v in the AFG in random order. To generate a new value for v, for each of its possible values we call FACTORSSCORE to compute the product of all the factors in the AFG that share an edge with v. This computation will re-run the checker for every check in which v appears. The result is a single non-negative score for each value d of v. The scores are then normalized to create a probability distribution over v's values, and finally a new value for v is sampled from this distribution.

After the burn-in completes, we take a snapshot of the values of *all* variables and store it as a single sample from P(A). From these samples we estimate probabilities for annotations. For example, to estimate the probability that an annotation has the value ro, we simply count the fraction of samples where it had the value ro.

This naïve description is subject to numerous well-known improvements. First, all computations are done in log-space to avoid problems with arithmetic roundoff. Second, we apply standard "annealing" tricks to improve convergence of the simulated Markov chain.

Our most important optimization (which we devised specifically for AFG inference), involves the execution of the checker. Because the checker will be executed *many* times, we cache checker results for each check by recording what values in *a* the checker consulted and storing the outcome of the check in a trie. This memoization is agnostic to the details of the checker itself, and leads to a two orders of magnitude speedup. The caches tend to be fairly well populated after generating 3-5 samples for the AFG, and is the primary reason the algorithm scales to analyzing real codebases.

6.3 Learning Parameters

If some annotations are known for a codebase, the parameters of an AFG can be learned (or tuned) by applying machine learning. The general approach to learning parameters for factor graphs is to apply gradient ascent to maximizing the likelihood function, where in this case the data is the set of known annotations. At a high level, gradient ascent iteratively tunes the parameters of the AFG to maximize the probability that the known annotations would be predicted using Equation 1. Full derivation of the gradient and the specific details for getting gradient ascent to work for AFGs is beyond the scope of this paper, but complete details can be found in [20]. Section 7.4 discusses our experience with parameter learning for leveraging codebase naming conventions.

		AF	AFG Size		Manually Classified Annotations					
Codebase L	ines (10 ³)	$ \mathbf{A} $	# Checks	ro	$\neg ro$	$\frac{ro}{\neg ro}$	co	$\neg co$	<u>co</u>	Total
SDL	51.5	843	577	35	25	1.4	16	31	0.51	107
OpenSSH	80.12	717	3416	45	28	1.6	10	108	0.09	191
GIMP	568.3	4287	21478	62	24	2.58	7	109	0.06	202
XNU	1381.1	1936	9169	35	49	0.71	17	99	0.17	200
Linux	6580.3	10736	92781	21	31	0.67	19	93	0.20	164

Table 3: Quantitative breakdown for each codebase of: (1) the size of the codebase, (2) the size of constructed AFG, and (3) the composition of the test set used for evaluating annotation accuracy. For the AFG statistics, |A| denotes the number of annotation variables.

7 Evaluation

We applied our technique to five open source codebases: SDL, OpenSSH, GIMP, XNU, and the Linux kernel. Table 3 gives the size of each codebase and its corresponding AFG. For each codebase's AFG we generated 100 samples using Gibbs sampling to estimate probabilities for annotations. We sort annotations from most-to-least probable based on these annotation probabilities. Note, we provided no "seed" annotations to the inference engine (e.g., did not include malloc, free). Since ro and co represent two distinct populations of annotations, we evaluate their accuracy separately. For our largest AFG (Linux), Gibbs sampling took approximately 13 hours on a 3 GHz dual-core Intel Xeon Mac Pro, while for the smallest AFG (SDL) sampling finished in under 5 minutes. Unless otherwise noted, the AFGs we evaluate include the factors for modeling prior biases (§ 3.4) and for multiple behavioral tests (§ 5.1).

Table 3 gives the breakdown of the "test set" of annotations we manually classified for each codebase. We selected the test set by: (1) automatically extracting a list of all functions involved in a check, (2) randomly picking n functions (100 $\leq n \leq$ 200) from this list, and (3) hand classifying these n functions. Note that this method produces a very harsh test set for inference because it picks functions with few callsites as readily as those with many. A seemingly innocent change produces a drastically easier test set: pick n functions from a list of all callsites. The selected test set would inflate our inference abilities since the probability of picking a function scales with the number of callsites it has. As a result, most functions in the test set would have many callsites, making inference much easier. In aggregate, we manually classified around 1,000 functions, giving a very comprehensive comparison set.

We first measure the accuracy of inferred annotations (§ 7.2). We then discuss the model's resilience to unanticipated coding idioms (§ 7.3). We next discuss our experience extending the core AFG model with keyword factors (§ 7.4). Finally we discuss using inferred annotations as safety nets for bug-finding tools (§ 7.5) and for finding bugs (§ 7.6).

7.1 Codebases

We evaluated our technique on important, real-world applications and systems based on both their size and their disparate implementations. We strove for project diversity in coding idioms and resource management.

The smallest project, SDL, is a challenge for inference because most functions have few callsites (most are called fewer than four times). Further, because it was originally developed for porting games from Windows to Linux, it employs uncommonly-used resource management functions from external libraries such as XLib. Thus, inferring these functions is not only challenging but also useful. As our results show, the AFG model readily infers correct ro and co annotations for functions in SDL that have as few as one or two callsites.

OpenSSH and the GIMP are widely-employed applications with modest to large source code size. The GIMP image manipulation program uses custom memory management functions and a plug-in infrastructure where plug-ins are loaded and unloaded on demand and where leaking plug-in memory is not considered an error because of their short lifetime. Despite such noise, our AFG worked well, and discovered several memory leaks in the GIMP core.

XNU (the kernel for Mac OS X) contains many specialized routines for managing kernel resources. Our results are mostly for the core kernel because much of the rest of XNU is written in C++, which our front-end does not handle. Inferred annotations for XNU immediately led to the discovery of bugs.

The Linux 2.6.10 kernel is a strong test because it is huge and its code frequently breaks the *ro-co* idiom due to sporadic use of (among other things) reference-counting and weird pointer mangling. Despite these challenges, our AFG successfully analyzed Linux with a reasonable annotation accuracy.

Note that AFG size scales with the number of checks, which only roughly correlates with code size: we ignore functions not involved in at least one check and (by necessity) skip function implementations our C front-end had difficultly parsing.

7.2 Annotation Accuracy

This section measures the accuracy of our inferred specifications. Our first experiment, shown in Table 4, gives a feel for the initial accuracy of inferred annotations. It presents the results from the first 10 and 20 inspections of the highest probability annotations for each codebase. These are selected from *all* the annotation variables in the AFG. The table assumes that the *ro* and *co* annotations are inspected separately, although this need not be the case in practice. The first few inspections are important as they represent our most confident annotations and will be the basis of a user's initial impression of the

		Inspections by $P(ro)$		Inspections by P(co	
Codebase	Inspections	ro's	¬ro's	co's	¬co's
SDL	10	10	0	9	1
	20	20	0	17	3
OpenSSH	10	10	0	10	0
	20	19	1	16	4
GIMP	10	10	0	9	1
	20	20	0	16	4
XNU	10	9	1	9	1
	20	16	4	17	3
Linux	10	8	2	9	1
	20	17	3	18	2

Table 4: Absolute number of ro(co) annotations found within the first 10 and 20 inspections for each codebase.

results. These top ranked annotations have near perfect accuracy for ro's and co's on all codebases.

We then more thoroughly measure annotation accuracy by comparing inferred annotations to the entire test set for each codebase (from Table 3). We also compare the accuracy of our base AFG against two ablated (i.e., broken) ones: AFG-NoFPP, which measures the effect of decreasing checker power, and AFG-RENAME, which measures the effect of decreased correlation.

Figure 7 shows the annotation accuracy of all three models for each codebase test set using Receiver Operating Characteristics (ROC) curves [11]. The ROC curve for ro annotations plots the classification accuracy as the classification probability threshold t slides from 1 to 0. Any annotation with a probability P(A = ro) > tis classified as ro, and $\neg ro$ otherwise. The x-axis depicts, for each value of t, the cumulative fraction of all the $\neg ro$'s in the test set that were mistakenly classified as ro (the false positive rate, or FPR). The y-axis depicts the cumulative fraction of all the ro's in the test set that were correctly classified as ro (the true positive rate, or TPR). Perfect annotation inference would yield a 100% TPR with a 0% FPR (visually a step, with a line segment from (0,0) to (0,1) and another from (0,1) to (1,1), as all the ro's appear before all of the $\neg ro$'s when the annotations are sorted by their probabilities. Random labeling yields a diagonal line from (0,0) to (1,1). With ROC curves we can easily compare accuracy across code bases since they are invariant to test set size and the relative skew of ro's to $\neg ro$'s and co's to $\neg co$'s.

Basic AFG: as can be seen in the figure, the AFG model has very high accuracy for all codebases except Linux, where accuracy is noticeably lower but still quite good. SDL, had both the least code and the highest annotation accuracy, with a nearly perfect accuracy for co's. Further, 35% of all ro's for SDL are found without inspecting a single $\neg ro$. For OpenSSH we observe around a 90% TPR with a FPR of 10% or less. Both GIMP and XNU observe an 80% or better TPR for both ro's and co's with a 20% FPR.

Accuracy appears to decrease with increased codebase size. While inspecting results, we observed that larger

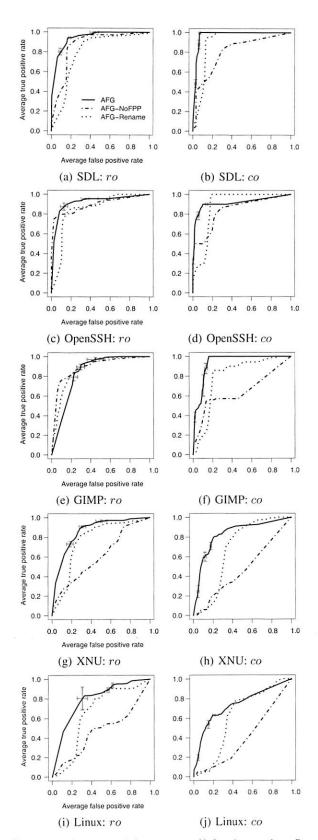


Figure 7: ROC curves depicting accuracy of inferred annotations. Results are averaged over 10 runs where we add Gaussian noise to AFG parameters. Error bars (shown at the classification probability thresholds of 0.95, 0.7, and 0.4) depict standard deviations.

	# Callsites per Function		% of Functions with		
Codebase	Mean	Median	≥ 5 Callsites	≥ 10 Callsites	
SDL	4.51	2	22.4	9.3	
OpenSSH	7.41	2	23.0	11.5	
GIMP	13.11	2	32.6	21.2	
XNU	9.19	2	24.0	11.5	
Linux	4.22	2	20.7	9.7	

Table 5: Summary statistics of the number of callsites per function (only for callsites that are consulted by at least one check). Since the mean number of callsites per function is much higher than the median, the number of callsites has a long-tailed distribution.

codebases frequently violate the ro-co idiom and have a lot of code beyond the reasoning power of our checker. However, even the hardest codebase, Linux, had decent accuracy: we were able to infer 70% (62%) of ro (co) functions with a 30% FPR.

While Figure 7 depicts overall codebase accuracy (with an aggregate annotation accuracy of 80-90% on the first four codebases, assuming a classification probability threshold of 0.5), in truth these numbers are highly pessimistic. First, as shown in Table 4, the accuracy of both ro and co annotations for the top ranked inspections (from the set of all inferred annotations) is near-perfect across all codebases. Thus, users will almost always see valid annotations when inspecting the highest confidence results. More importantly, we observe on all codebases that the distribution of the number of callsites per function is highly skewed (Table 5), with a little over 50% of all functions being called at most twice, and only 20-30% being called five times or more. Assuming we classify, using a probability threshold of 0.5, the annotations for functions that are called five or more times, we observe on all codebases (including Linux) an 80-95% accuracy for both ro and co annotations, and an overall accuracy of 90%. While annotation accuracy for single callsite functions is important, correctly inferring annotations for functions that are called at least a few times often has much higher practical impact because those annotations will be used throughout a codebase (and often provide the greatest traction for bug-finding tools). Moreover, our annotation accuracy for single callsite functions is far better than random labeling. For such functions, on Linux the AFG model correctly infers around twice as many ro and co annotations as random labeling for the same number of inspected false positives.

AFG-NoFPP: reduces the checker's analysis precision by disabling false path pruning [6] (FPP), which eliminates many bogus paths through the code that static analysis would otherwise believe existed. In practice FPP reduces false error messages by recognizing many control-dependent branches such as:

if(
$$x == 1$$
) $p = malloc();$...

if(
$$x == 1$$
) free(p);

Disabling FPP will cause the checker to believe that there are four possible paths instead of two. Consequently, removing FPP generates significant noise in the checker's results (the next section provides more detail). Unsurprisingly, for all codebases, AFG-NoFPP has significantly reduced accuracy for co annotations due to false paths. The main way false paths cause problems for ro-co inference is that they make it appear that some functions claim a resource when in reality they are never called in this manner. While this always has a negative impact on co accuracy, because of the large ratio of ro's to $\neg ro$'s for GIMP (Table 3) this over-inflates the evidence for ro annotations and leads to increased ro accuracy while still having poor co accuracy. AFG-NoFPP performs very poorly on XNU and Linux (to the degree that random labeling does better) as the use of many resources in these codebases are highly control-dependent.

AFG-Rename: evaluates the benefit of exploiting inter-correlation between annotations by systematically destroying it. We do so by "renaming," for each callsite, the called function so that each callsite refers to a distinct function. This causes AFG-Rename to have an annotation for each callsite of a function. We compute probabilities for these annotations as normal, but then compute probabilities for the original function annotations (nonrenamed) by averaging the probabilities across the percallsite annotations. The end result is that by we can see how much correlation helps inference by contrasting the performance of AFG-Rename to AFG.

The curves for AFG-RENAME perform closest to AFG on the smallest codebases, and gradually diverges (especially for *co* accuracy) as we look at larger codebases. This is largely due to the increased amount of correlation in the larger codebases. The accuracy of AFG-RENAME diverges significantly from AFG for codebases larger than OpenSSH so that *co* performance degenerates to that produced by AFG-NoFPP. (Note that for OpenSSH and GIMP, an apparent bump in *co* accuracy of AFG-RENAME over AFG in the tail of the ROC curve is due a single *co* annotation, and is within the margin of noise induced by the test set size.)

Sensitivity to magic numbers. As part of this experiment we also measured our earlier claim that even rough guesses of AFG parameters generate acceptable (initial) results by doing the following sensitivity analysis. We perturb each AFG parameter by a randomly generated amount of Gaussian noise ($\sigma^2 = 0.02$), which maintains the relative ordering between parameter values, but skews their values slightly and thus their relative odds. We generate 10 sets in this manner, use them to infer annotations, and then report averages and standard deviations (depicted as error bars in Figure 7) across all runs. In general, as the figure shows, the error bars for AFG are quite small, illustrating that our inference re-

sults were robust to small perturbations in parameter values. While we believe the parameters are amendable to tuning, the choice of numbers is not so brittle as to cause violent changes in results when perturbed slightly.

7.3 Detecting Unanticipated Coding Idioms

Initially, classifications for Linux were slightly worse than the other codebases because its ownership model is more subtle than the one we attempt to infer. Using the fit model variables discussed in Section 5.3, we quickly identified a corner case in Linux that we needed to explicitly model in our checker. A common practice in Linux is to conflate the values of pointers and to store error codes in them instead of memory addresses. Since the kernel resides in a restricted address space, error codes can be mangled into pointer values that lie outside this region. Such pointers are frequently tested as follows:

$$\begin{array}{ll} p \ = \ foo(\ \dots\); \\ \textbf{if}(\ IS_ERR_PTR(p)\)\ \big\{\ /*\ \textit{error\ path}\ \ */\ \big\} \end{array}$$

On the true branch of such tests the reference to p appears to be lost, when in reality it was not a valid pointer. This causes serious problems because IS_ERR_PTR is far outside our simple ro-co model. Because it appears hundreds of times, inference computed a probability of 0.99 that this function did not fit ($\neg fit$) the ro-co model and all checks involving IS_ERR_PTR were effectively removed as evidence. We immediately noticed the highly probable $\neg fit$ value for IS_ERR_PTR and modified our checker to recognize the function and prune analysis of paths where its return value for a tracked pointer is true, allowing us to glean evidence from these checks.

7.4 Additional Information

Our last experiment for annotation accuracy looks at how overall annotation accuracy for XNU improves as inference is given additional information. We use (1) a set of 100 known good function annotations to serve as a training set — in practice these would be harvested as a checking tool is repeatedly run over a code base, and (2) a list of substrings a programmer feels might be relevant to the labeling of a function. We provided a relatively small set of 10 strings such as "alloc" and "get," but there is nothing to keep a motivated user from listing all interesting strings from their problem domain. We applied parameter learning to train the parameters for the keyword factors based on a training set (as described in [20]), and tested the classification accuracy on the remaining 100. We set the classification probability threshold at 0.5 to indicate whether an annotation was ro or $\neg ro$ (co or $\neg co$) to get a measure of overall accuracy.

For XNU, the baseline aggregate accuracy (inference without knowing the training set) was 81.2%, and with the addition of knowing the annotations in the training set

	Function	Param.	Label	Sites	Prob.
	XAllocWMHints	ret	ro	1	0.99
X11	XFree	I	co	9	0.98
API	XGetVisualInfo	ret	ro	1	0.97
	XListPixmapFormats	ret	ro	1	0.97
	X11_CreateWMCursor	ret	ro	1	0.95
	XOpenDisplay	ret	ro	2	0.86
	XGetModifierMapping	ret	ro	1	0.86
	XCreateGC	ret	ro	2	0.84
	XFreeGC	2	co	2	0.82
	XFreeModifierMap	1	co	1	0.79
	XCloseDisplay	1	co	2	0.76
	dlopen	ret	ro	1	0.95
C	opendir	ret	ro	1	0.87
Standard	setmntent	ret	ro	1	0.74
Library	closedir	1	co	1	0.73
	endmntent	1	co	1	0.58

Table 6: A selection of correctly inferred labels for external functions inferred from analyzing SDL and not in the Coverity Prevent "root set." "Sites" is the number of callsites for the given function utilized by the program analysis and used for inference.

accuracy of the test set increased to 84.2%. Knowledge of the training set during inference simulates already knowing some of the annotations. Equipped with the keyword information alone accuracy was 89.1%, with the addition of knowing the annotations in the training set the accuracy was 90.1%. This experiment demonstrates the power of our technique: we are able to easily incorporate additional information and have that information improve the accuracy of our results. We also benchmarked these results against an AFG that only included keyword and prior belief factors. While the top ranked ro and co inferred annotations from this model were usually correct, very quickly accuracy degrades as annotations are inspected. Overall, ro and co accuracy is 22-33% worse when using keyword information alone.

7.5 Safety Nets for Bug-Finding Tools

We evaluate our classifications by comparing the inferred ro and co functions classifications for SDL against the allocator and deallocator functions used by Prevent [7].

Coverity Prevent is a commercial static analysis tool that contains several analyses to detect resource errors. It performs an unsound relaxation analysis through the call-graph to identify functions that transitively call "root" allocators and deallocators such as malloc and free. Prevent's analysis is geared to find as many defects as possible with a low false error rate. The set of allocators and deallocators yielded by our inference and Prevent (respectively) perform a synergistic cross-check. Prevent will miss some allocators because they do not transitively call known allocators, and static analysis imprecision may inhibit the diagnosis of some functions that do. On the other hand, Prevent can classify some functions we miss since it analyzes more code than we do, has better alias tracking and better path-sensitivity. It found four classifications that our inference missed. We inspected each of the four and all were due to the fact

```
void gimp_enum_stock_box_set_child_padding (...) {
   GList *list;
   ...
   for( list = gtk_container_get_children(...); list;
        list = g_list_next(list) ) { ... }
}
```

Figure 8: [BUG] gtk_container_get_children returns a newly allocated list. The pointer to the head of the list is lost after the first iteration of the loop.

Figure 9: Source code from the Gtk+ library of gtk_container_get_children. The function performs a complicated copy and reversal of a linked list. Inference labels the return value (correctly) as ro without analyzing the implementation.

that our static analysis made mistakes rather than a flaw in the inference algorithm.

Our belief that manual specifications will have holes was born out. Inference found over 40 allocator and deal-locator functions that Prevent missed. Table 6 gives a representative subset. Prevent missed the bulk of these because SDL uses obscure interfaces which were not annotated and which were not part of the SDL source code (and therefore it could not analyze them). It is unclear what caused some of the omissions, though analysis mistakes are the plausible candidate. This experiment shows that even highly competent, motivated developers attempting to annotate all relevant functions in a "root set" easily miss many candidates.

Finally, our inference found an annotation misclassified by Prevent's relaxation. The function SDL_ConvertSurface returns a pointer obtained by calling SDL_CreateRGBSurface. This function, in turn, returns a pointer from malloc, a well-known allocator function. Prevent misclassifies the return value of SDL_ConvertSurface as ¬ro, an error likely due to the complexity of these functions. Our checker also had problems understanding the implementation of these functions, but correctly inferred an ro annotation for its return value based on the context of how SDL_ConvertSurface was used. We reported this case to Coverity developers and they confirmed it was a misclassified annotation.

7.6 Defect Accuracy

Using our inference technique, we diagnosed scores of resource bugs (most of them leaks) in all five codebases, but since our focus was on annotation accuracy, we did not perform an exhaustive evaluation on all codebases of the quantity of bugs our tool found. All diagnosed bugs were discovered by ranking errors by the probabilities of the annotations involved, and for each codebase led to the discovery of bugs within minutes.

One particular bug found in GIMP highlights the inferential power of AFGs. Figure 8 shows an incorrect use of the function gtk_container_get_children (whose return value we correctly annotate as ro) in the core GIMP code and illustrates the power of being able to infer the annotation for a function based on the context in which it is used. This function returns a freshly allocated linked list, but in this code fragment a list iteration is performed and the head of the list is immediately lost. We did not analyze the source of the Gtk+ library when analyzing GIMP; this annotation was inferred solely from how the function was used.

The implementation of gtk_container_get_children, excerpted in Figure 9, shows how the list is created by performing a complicated element-wise copy (involving a custom memory allocator) after which the list is reversed, with the new head being the "owning" pointer of the data structure. Even if the implementation were available, understanding this function would pose a strenuous task for classic program analysis.

8 Related Work

Most existing work on inferring specifications from code looks for rules of the form "do not perform action a before action b" or "if action a is performed then also perform action b." The inferred rules are captured in (probabilistic) finite-state automata (FSAs and PFSAs).

Engler et al [9] infer a variety of properties from template rules, including *a-b* pairs in systems code. Examples include inferring whether malloc is paired with free, lock with unlock, whether or not a null pointer check should always be performed on the return value of a function, etc. The intuition is that frequently occurring patterns are likely to be rules, while deviant behavior of strongly observed patterns are potential bugs. Our work can be viewed as a natural generalization of this earlier work to leverage multiple sources of information and exploit correlation.

Weimer and Necula [30] observed that API rule violations occur frequently on "error paths" such as exception handling code in Java programs. Consequently, they weight observations on these paths differently from regular code. We observe similar mistakes in systems code, although there identifying an error path is not always trivial. This poses a potential form of *indirect* correlation to exploit for inferring annotations. High confidence specifications can be used to infer error paths, which in turn can be used to infer other specifications.

Li and Zhou [22] and Livshits and Zimmerman [25] look for generalized *a-b* patterns in code. Li and Zhou look for patterns across a codebase, whereas Livshits and Zimmerman look at patterns that are correlated through version control edits. While general, these approaches are based on data mining techniques that often require large amounts of data to derive patterns, meaning it is unlikely they will be able to say anything useful about infrequently called functions. It is also unclear how to extend them to incorporate domain specific knowledge.

Ammons et al [2] have a dynamic analysis to learn probabilistic finite-state automatons (PFSAs) that describe the dependency relationship between a series of function calls with common values passed as arguments. Concept analysis is then used to aid the user, somewhat successfully, in the daunting task of debugging the candidate PFSAs [3]. Their method suffers from the usual code coverage hurdles inherent in a run-time analysis, making it difficult for such methods to infer properties about rarely executed code. They also assume program traces that illustrate perfect compliance of the rules being inferred (i.e., all traces are "bug-free"). This limitation is overcome by Yang et al [34], but unlike AFGs, their mechanism for handling noise and uncertainty is specific to the patterns they infer. In addition, it is unclear how to extend either method to incorporate domain specific knowledge in a flexible and natural way.

Whaley et al [31] derive interface specifications of the form "a must not be followed by b" for Java method calls. Their technique relies on static analysis of a method's implementation to find out if calling b after a would produce a runtime error. Alur et al [1] extend their method using model checking to handle sequences of calls σ and to provide additional soundness. Although powerful, these techniques examine the implementation of methods rather than the context in which they are used. The results of such techniques are thus limited by the ability of the analysis to reason about the code, and may not be able to discover some of the indirectly correlated specifications our approach provides.

Hackett et al [14] partially automate the task of annotating a large codebase in order to find buffer overflows. Their method is used in an environment with significant user input (over a hundred thousand user annotations) and is specific to their problem domain. Further, it is unclear how to extend their technique to leverage additional behavioral signatures.

Ernst and his collaborators developed Daikon [10,29], a system that infers program invariants via run-time monitoring. Daikon finds simple invariants specifying relational properties such as $a \geq b$ and $x \neq 0$, although

conceivably inferring other properties is possible. Their method is not statistical, and invariants inferred require perfect compliance from the observed program.

More distantly related are techniques that tackle the inverse problem of "failure inference" for postmortem debugging. The goal is to diagnose the cause of a failstop error such as an assertion failure or segmentation fault. Both pure static analysis [26] and statistical debugging techniques have been employed with inspiring results [16, 23, 24]. Although specification inference and failure inference have different goals, we believe that many of the ideas presented in this paper could be readily applied in that domain.

9 Conclusion

This paper presented a novel specification inference framework based on factor graphs whose key strength is the ability to combine disparate forms of evidence, such as those from behavioral signatures, prior beliefs, and ad hoc knowledge, and reason about them in the common currency of factors and probabilities. We evaluated the approach for the ownership problem, and achieved high annotation accuracy across five real-world codebases. While our checker was primitive, with inferred annotations we immediately discovered numerous bugs, including those that would impossible to discover with state-of-the-art program analysis alone.

10 Acknowledgements

This research was supported by NSF grants CNS-0509558, CCF-0326227-002, NSF CAREER award CNS-0238570-001, and the Department of Homeland Security. We would like to thank Andrew Sakai and Tom Duffy for their help in sorting out XNU bugs. We are grateful to our shepherd Peter Druschel for his help, and Cristian Cadar, Junfeng Yang, Can Sar, Peter Pawlowski, Ilya Shpitser, and Ashley Dragoman for their helpful comments.

References

- R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2005.
- [2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 4–16, New York, NY, USA, 2002.
- [3] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus. Debugging temporal specifications with concept analysis. In PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pages 182–195, New York, NY, USA, 2003. ACM Press.
- [4] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. Software: Practice and Experience, 30(7):775–802, 2000.
- [5] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Com*puter and communications security, pages 235 – 244. ACM Press, 2002.
- [6] A. Chou. Static Analysis for Finding Bugs in Systems Software. PhD thesis, Stanford University, 2003.
- [7] Coverity Prevent. http://www.coverity.com.
- [8] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, June 2002.

- [9] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In Eighteenth ACM Symposium on Operating Systems Principles, 2001.
- [10] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *IEEE Trans*actions on Software Engineering, Feb. 2001.
- [11] T. Fawcett. ROC Graphs: Notes and practical considerations for data mining researchers. Technical Report HPL-2003-4, Intelligent Enterprise Technologies Laboratory, HP Laboratories Palo Alto, January 2003.
- [12] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *PLDI* 2002, pages 234–245, 2002.
- [13] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter, editors. Markov Chain Monte Carlo in Practice. Chapman and Hall/CRC, 1996.
- [14] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, ICSE, pages 232–241. ACM, 2006.
- [15] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, 2002.
- [16] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, May 2002.
- [17] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In Proceedings of the Winter USENIX Conference, Dec. 1992.
- [18] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In ACM SIGPLAN 2003 conference on Programming language design and implementation, 2003.
- [19] G. E. Hinton. Products of experts. Technical report, Gatsby Computational Neuroscience Unit, University College London.
- [20] T. Kremenek, A. Ng, and D. Engler. A factor graph model for software bug finding. Technical report, Stanford University, 2006.
- [21] F. R. Kschischang, B. J. Frey, and H. A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 2001.
- [22] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Sept. 2005.
- [23] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, CA USA, 2003.
- [24] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM SIG-PLAN conference on Programming language design and implementation*, pages 15–26, New York, NY, USA, 2005. ACM Press.
- [25] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In 13th ACM SIGSOFT international symposium on Foundations of software engineering, pages 296–305, New York, NY, USA, 2005. ACM Press.
- [26] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: explaining program failures via postmortem static analysis. In Proceedings of the 12th ACM SIGSOFT rwelfth international symposium on Foundations of software engineering, pages 63–72, New York, NY, USA, 2004.
- [27] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In Proceedings of Conference on Compilier Construction, 2002.
- [28] N. Nethercote. Dynamic Binary Analysis and Instrumentation. PhD thesis, University of Cambridge, 2004.
- [29] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, 2004.
- [30] W. Weimer and G. Necula. Mining temporal specifications for error detection. In 11th International Conference on Tools and Algorithms For The Construction And Analysis Of Systems, 2005.
- [31] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of objectoriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT* international symposium on Software testing and analysis, 2002.
- [32] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In 13th ACM SIGSOFT international symposium on Foundations of software engineering, pages 115–125, New York, NY, USA, 2005.
- [33] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 351–363, New York, NY, USA, 2005.
- [34] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In 28th international conference on Software engineering, pages 282–291, New York, NY, USA, 2006.
- [35] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an extensible, expressive system and language for statically checking security properties. In 10th ACM conference on Computer and communications security, 2003.
- [36] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. In Exploring Artificial Intelligence in the New Millennium, pages 239–269. Morgan Kaufmann Publishers Inc., 2003.

HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance

James Cowling¹, Daniel Myers¹, Barbara Liskov¹, Rodrigo Rodrigues², and Liuba Shrira³

¹MIT CSAIL, ²INESC-ID and Instituto Superior Técnico, ³Brandeis University

{cowling, dsm, liskov, rodrigo, liuba}@csail.mit.edu

Abstract

There are currently two approaches to providing Byzantine-fault-tolerant state machine replication: a replica-based approach, e.g., BFT, that uses communication between replicas to agree on a proposed ordering of requests, and a quorum-based approach, such as Q/U, in which clients contact replicas directly to optimistically execute operations. Both approaches have shortcomings: the quadratic cost of inter-replica communication is unnecessary when there is no contention, and Q/U requires a large number of replicas and performs poorly under contention.

We present HQ, a hybrid Byzantine-fault-tolerant state machine replication protocol that overcomes these problems. HQ employs a lightweight quorum-based protocol when there is no contention, but uses BFT to resolve contention when it arises. Furthermore, HQ uses only 3f+1 replicas to tolerate f faults, providing optimal resilience to node failures.

We implemented a prototype of HQ, and we compare its performance to BFT and Q/U analytically and experimentally. Additionally, in this work we use a new implementation of BFT designed to scale as the number of faults increases. Our results show that both HQ and our new implementation of BFT scale as f increases; additionally our hybrid approach of using BFT to handle contention works well.

1 Introduction

Byzantine fault tolerance enhances the availability and reliability of replicated services in which faulty nodes may behave arbitrarily. In particular, state machine replication protocols [9, 19] that tolerate Byzantine faults allow for the replication of any deterministic service.

Initial proposals for Byzantine-fault-tolerant state machine replication [18, 2] relied on all-to-all communication among replicas to agree on the order in which to

execute operations. This can pose a scalability problem as the number of faults tolerated by the system (and thus the number of replicas) increases.

In their recent paper describing the Q/U protocol [1], Abd-El-Malek et al. note this weakness of agreement approaches and show how to adapt Byzantine quorum protocols, which had previously been mostly limited to a restricted read/write interface [12], to implement Byzantine-fault-tolerant state machine replication. This is achieved through a client-directed process that requires one round of communication between the client and the replicas when there is no contention and no failures.

However, Q/U has two shortcomings that prevent the full benefit of quorum-based systems from being realized. First, it requires a large number of replicas: 5f+1 are needed to tolerate f failures, considerably higher than the theoretical minimum of 3f+1. This increase in the replica set size not only places additional requirements on the number of physical machines and the interconnection fabric, but it also increases the number of possible points of failure in the system. Second, Q/U performs poorly when there is contention among concurrent write operations: it resorts to exponential back-off to resolve contention, leading to greatly reduced throughput. Performance under write contention is of particular concern, given that such workloads are generated by many applications of interest (e.g. transactional systems).

This paper presents the Hybrid Quorum (HQ) replication protocol, a new quorum-based protocol for Byzantine fault tolerant systems that overcomes these limitations. HQ requires only 3f+1 replicas and combines quorum and agreement-based state machine replication techniques to provide scalable performance as f increases. In the absence of contention, HQ uses a new, lightweight Byzantine quorum protocol in which reads require one round trip of communication between the client and the replicas, and writes require two round trips. When contention occurs, it uses the BFT state machine replication algorithm [2] to efficiently order the contend-

ing operations. A further point is that, like Q/U and BFT, HQ handles Byzantine clients as well as servers.

The paper additionally presents a new implementation of BFT. The original implementation of BFT [2] was designed to work well at small f; our new implementation is designed to scale as f grows.

The paper presents analytic results for HQ, Q/U, and BFT, and performance results for HQ and BFT. Our results indicate that both HQ and the new implementation of BFT scale acceptably in the region studied (up to f=5) and that our approach to resolving contention provides a gradual degradation in performance as contention rises.

The paper is organized as follows. Section 2 describes our assumptions about the replicas and the network connecting them. Section 3 describes HQ, while Section 4 describes a number of optimizations and our new implementation of BFT. Section 5 presents analytic results for HQ, BFT, and Q/U performance in the absence of contention, and Section 6 provides performance results for HQ and BFT under various workloads. Section 7 discusses related work, and we conclude in Section 8.

2 Model

The system consists of a set $\mathcal{C} = \{c_1, ..., c_n\}$ of client processes and a set $\mathcal{R} = \{r_1, ..., r_{3f+1}\}$ of server processes (or replicas). Client and server processes are classified as either correct or faulty. Correct processes are constrained to obey their specification, i.e., they follow the prescribed algorithms. Faulty processes may deviate arbitrarily from their specification: we assume a Byzantine failure model [8]. Note that faulty processes include those that fail benignly as well as those suffering from Byzantine failures.

We assume an asynchronous distributed system where nodes are connected by a network that may fail to deliver messages, delay them, duplicate them, corrupt them, or deliver them out of order, and there are no known bounds on message delays or on the time to execute operations. We assume the network is fully connected, i.e., given a node identifier, any other node can (attempt to) contact the former directly by sending it a message.

For liveness, we require only that if a client keeps retransmitting a request to a correct server, the reply to that request will eventually be received, plus the conditions required for liveness of the BFT algorithm [2] that we use as a separate module.

We assume nodes can use unforgeable digital signatures to authenticate communication. We denote message m signed by node n as $\langle m \rangle_{\sigma_n}$. No node can send $\langle m \rangle_{\sigma_n}$ (either directly or as part of another message) on the network for any value of m, unless it is repeating a message that has been sent before or it knows n's private

key. We discuss how to avoid the use of computationally expensive digital signatures in Section 3.3. Message Authentication Codes (MACs) are used to establish secure communication between pairs of nodes, with the notation $\langle m \rangle_{\mu_{xy}}$ indicating a message authenticated using the symmetric key shared by x and y. We assume a trusted key distribution mechanism that provides each node with the public key of any other node in the system, thus allowing establishment of symmetric session keys for use in MACs.

We assume the existence of a collision-resistant hash function, h, such that any node can compute a digest h(m) of message m and it is impossible to find two distinct messages m and m' such that h(m) = h(m').

To avoid replay attacks we tag certain messages with nonces that are signed in replies. We assume that when clients pick nonces they will not choose a repeated value.

3 HQ Replication

HQ is a state machine replication protocol that can handle arbitrary (deterministic) operations. We classify operations as *reads* and *writes*. (Note that the operations are not restricted to simple reads or writes of portions of the service state; the distinction made here is that read operations do not modify the service state whereas write operations do.) In the normal case of no failures and no contention, write operations require two phases to complete (we call the phases write-1 and write-2) while reads require just one phase. Each phase consists of the client issuing an RPC call to all replicas and collecting a quorum of replies.

The HQ protocol requires 3f+1 replicas to survive f failures and uses quorums of size 2f+1. It makes use of *certificates* to ensure that write operations are properly ordered. A certificate is a quorum of authenticated messages from different replicas all vouching for some fact. The purpose of the write-1 phase is to obtain a *time-stamp* that determines the ordering of this write relative to others. Successful completion of this phase provides the client with a certificate proving that it can execute its operation at timestamp t. The client then uses this certificate to convince replicas to execute its operation at this timestamp in the write-2 phase. A write concludes when 2f+1 replicas have processed the write-2 phase request, and the client has collected the respective replies.

In the absence of contention, a client will obtain a usable certificate at the end of the write-1 phase and succeed in executing the write-2 phase. Progress is ensured in the presence of slow or failed clients by the writeBackWrite and writeBackRead operations, allowing other clients to complete phase 2 on their behalf. When there contention exists, however, a client may not obtain a usable write certificate, and in this case it asks

the system to *resolve* the contention for the timestamp in question. Our contention resolution process uses BFT to order the contending operations. It guarantees

- 1. if the write-2 phase completed for an operation o at timestamp t, o will continue to be assigned to t.
- if some client has obtained a certificate to run o at t, but o has not yet completed, o will run at some timestamp ≥ t.

In the second case it is possible that some replicas have already acted on the write-2 request to run o at t and as a result of contention resolution, they may need to undo that activity (since o has been assigned a different timestamp). Therefore all replicas maintain a single backup state so that they can undo the last write they executed. However, this undo is not visible to end users, since they receive results only when the write-2 phase has completed, and in this case the operation retains its timestamp.

3.1 System Architecture

The system architecture is illustrated in Figure 1. Our code runs as proxies on the client and server machines: the application code at the client calls an operation on the client proxy, while the server code is invoked by the server proxy in response to client requests. The server code maintains replica state; it performs application operations and must also be able to undo the most recently received (but not yet completed) operation (to handle reordering in the presence of contention).

The replicas also run the BFT state machine replication protocol [2], which they use to resolve contention; note that BFT is not involved in the absence of contention.

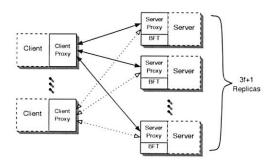


Figure 1: System Architecture

3.2 Normal Case

We now present the details of our protocol for the case where there is no need to resolve contention; Section 3.3

describes contention resolution. We present an unoptimized version of the protocol; optimizations are discusses in Section 4.

The system supports multiple objects. For now we assume that each operation concerns a single object; we discuss how to extend this to multi-object transactions in Section 3.6. Writers are allowed to modify different objects in parallel but are restricted to having only one operation outstanding on a particular object at a time. Writers number their requests on individual objects sequentially, which allows us to avoid executing modification operations more than once. A writer can query the system for information about its most recent write at any time.

A write certificate contains a quorum of grants, each of form $\langle cid, oid, op\#, h_{op}, vs, ts, rid \rangle_{\sigma_r}$, where each grant is signed by its replica r with id rid. A grant states that the replica has granted the client with id cid the right to run the operation numbered op# whose hash is h_{op} on object oid at timestamp ts. A write certificate is valid if all the grants are from different replicas, are correctly authenticated, and are otherwise identical. We use the notation c.cid, c.ts, etc., to denote the corresponding components of write certificate c. The vs component is a viewstamp; it tracks the running of BFT to perform contention resolution. A certificate C1 is later than certificate C2 if C1's viewstamp or timestamp is larger than that of C2. (A viewstamp is a pair consisting of the current BFT view number, plus the number assigned by BFT to the operation it executed most recently, with the obvious ordering.)

3.2.1 Processing at the Client

Write Protocol. As mentioned, writing is performed in two phases. In the first phase the writer obtains a certificate that allows it to execute the operation at a particular timestamp in the second phase.

Write-1 phase. The client sends a (WRITE-1, cid, oid, op#, op \rangle_{σ_c} request to the replicas. The following replies are possible:

- \(\partial\) WRITE-1-OK, grantTS, currentC\(\rangle\), if the replica granted the next timestamp to this client.
- $\langle WRITE-1-REFUSED, grantTS, cid, oid, op\#, currentC \rangle_{\mu_{cr}}$, if the replica granted the timestamp to some other client; the reply contains the grant to that client, plus the information about this client's request (to prevent replays).
- (WRITE-2-ANS, result, currentC, $rid\rangle_{\mu_{cr}}$, if the client's write has already been executed (this can happen if this client is slow and some other client performs the write for it see step 2 below).

In a WRITE-1-OK or WRITE-1-REFUSED reply, *currentC* is the certificate for the latest write done at that replica.

The client discards invalid replies; it processes valid replies as follows:

- If it receives a quorum of OKs for the same viewstamp and timestamp, it forms a write certificate from the grants in the replies and moves to the write-2 phase.
- 2. If it receives a quorum of refusals with the same viewstamp, timestamp, and hash, some other client has received a quorum of grants and should be executing a write phase 2. To facilitate progress in the presence of slow or failed clients, the client forms a certificate from these grants and performs the write followed by a repeat of its own WRITE-1 request: it sends a (WRITEBACKWRITE, writeC, w1) to the replicas, where w1 is a copy of its WRITE-1 request; replicas reply with their responses to the WRITE-1.
- 3. If it receives grants with different viewstamps or timestamps, it also performs a WRITEBACKWRITE, this time using the latest write certificate it received. This case can happen when some replica has not heard of an earlier write. Writebacks are sent only to the slow replicas.
- 4. If it receives a WRITE-2-ANS, it uses the certificate in the WRITE-2-ANS to move to phase 2. This case can happen if some other client performed its write (did step 2 above).
- 5. If it receives a quorum of responses containing grants with the same viewstamp and timestamp but otherwise different, it sends a RESOLVE request to the replicas; the handling of this request is discussed in Section 3.3. This situation indicates the possibility of write contention, The replies to the RESOLVE request are identical to replies to a WRITE-1 request, so the responses are handled as described in this section.

Write-2 phase. The client sends a $\langle WRITE-2, writeC \rangle$ request, where writeC is the write certificate it obtained in phase 1. Then it waits for a quorum of valid matching responses of the form $\langle WRITE-2-ANS, result, currentC, rid \rangle_{\mu_{cr}}$; once it has these responses it returns to the calling application. It will receive this many matching responses unless there is contention; we discuss this case in Section 3.3.

Read Protocol. The client sends $\langle \text{READ}, cid, oid, op, nonce \rangle_{\mu_{cr}}$ requests to the replicas. The nonce is used to uniquely identify the request, allowing a reader to distinguish the respective reply from a replay. The response to this request has the form $\langle \text{READ-ANS}, result, nonce, currentC, rid \rangle_{\mu_{cr}}$.

The client waits for a quorum of valid matching replies and then returns the result to the calling application. If it receives replies with different viewstamps or timestamps, it sends a $\langle \text{WRITEBACKREAD}, \textit{writeC}, \textit{cid}, \textit{oid}, \textit{op}, \textit{nonce} \rangle_{\mu_{cr}}$ to the (slow) replicas, requesting that they perform the write followed by the read. Here writeC is the latest write certificate received in the replies. This case can occur when a write is running concurrently with the read.

3.2.2 Processing at the Replicas

Now we describe processing at the replicas in the absence of contention resolution. Each replica keeps the following information for each object:

- *currentC*, the certificate for the current state of the object.
- grantTS, a grant for the next timestamp, if one exists
- ops, a list of WRITE-1 requests that are currently under consideration (the request that was granted the next timestamp, and also requests that have been refused), plus the request executed most recently.
- oldOps, a table containing, for each client authorized to write, the op# of the most recently completed write request for that client together with the result and certificate sent in the WRITE-2-ANS reply to that request.
- vs, the current viewstamp (which advances each time the system does contention resolution).

A replica discards invalid requests (bad format, improperly signed, or invalid certificate). It processes valid requests as follows:

Read request $\langle \text{READ}, cid, oid, op, nonce \rangle_{\mu_{cr}}$. The replica does an upcall to the server code, passing it the op. When this call returns it sends the result to the client in a message $\langle \text{READ-ANS}, result, nonce, currentC, rid \rangle_{\mu_{cr}}$. The nonce is used to ensure that the answer is not a replay.

Write 1 request $\langle \text{WRITE-1}, cid, oid, op\#, op} \rangle_{\sigma_c}$. If op# < oldOps[cid].op#, the request is old and is discarded. If op# = oldOps[cid].op#, the replica returns a WRITE-2-ANS response containing the result and certificate stored in oldOps[cid]. If the request is stored in ops, the replica responds with its previous WRITE-1-OK or WRITE-1-REFUSED response. Otherwise the replica appends the request to ops. Then if grantTS = null, it sets $grantTS = \langle c, oid, op\#, h, vs, currentC.ts+1, rid \rangle_{\sigma_r}$, where h is the hash of $\langle cid, oid, op\#, op \rangle$ and replies $\langle \text{WRITE-1-OK}, grantTS, currentC \rangle$; otherwise it

replies $\langle WRITE-1-REFUSED, grantTS, cid, oid, op\#, currentC \rangle_{\mu_{cr}}$ (since some other client has been granted the timestamp).

Write 2 request $\langle \text{WRITE-2}, writeC \rangle$. Any node is permitted to run a WRITE-2 request; the meaning of the request depends only on the contained certificate rather than on the identity of the sender. The certificate identifies the client c that ran the write-1 phase and the oid and op# it requested. The replica uses the oldOps entry for c to identify old and duplicate write-2 requests; it discards old requests and returns the write-2 response stored in oldOps[c] for duplicates.

If the request is new, the replica makes an upcall to the server code to execute the operation corresponding to the request. A replica can do the upcall to execute a WRITE-2 request only if it knows the operation corresponding to the request and it is up to date; in particular its vs = writeC.vs and currentC.ts = writeC.ts - 1. If this condition isn't satisfied, it obtains the missing information from other replicas as discussed in Sections 3.3 and 3.4, and makes upcalls to perform earlier operations before executing the current operation.

When it receives the result of the upcall, the replica updates the information in the *oldOps* for c, sets grantTS to null, sets ops to contain just the request being executed, and sets currentC = writeC. Then it replies $\langle WRITE-2-ANS, result, currentC, rid \rangle_{ucr}$.

WriteBackWrite and WriteBackRead. The replica performs the WRITE-2 request, but doesn't send the reply. Then it processes the READ or WRITE-1 and sends that response to the client.

3.3 Contention Resolution

Contention occurs when several clients are competing to write at the same timestamp. Clients notice contention when processing responses to a WRITE-1 request, specifically case (5) of this processing, where the client has received conflicting grants for the same viewstamp and timestamp. Conflicting grants normally arise because of contention but can also occur because a faulty client has sent different requests to different replicas.

In either case a client requests contention resolution by sending a $\langle \text{RESOLVE}, conflictC, w1 \rangle$ request to the replicas, where conflictC is a conflict certificate formed from the grants in the replies and w1 is the WRITE-1 request it sent that led to the conflict being discovered. The processing of this request orders one or more of the contending requests and performs those requests in that order; normally all contending requests will be completed.

To resolve contention we make use of the BFT state machine protocol [2], which is also running at the replicas. One of these replicas is acting as the BFT primary, and the server proxy code tracks this information, just like a client of BFT. However in our system, we use BFT only to reach agreement on a deterministic ordering of the conflicting updates.

3.3.1 Processing at Replicas

To handle contention, a replica has additional state:

- *conflictC*, either *null* or the conflict certificate that started the conflict resolution.
- backupC, containing the previous write certificate (for the write before the one that led to the currentC certificate).
- prev, containing the previous information stored in oldOps for the client whose request was executed most recently.

A replica that is processing a resolve request has a non-null value in *conflictC*. Such a replica is *frozen*: it does not respond to client write and resolve requests, but instead delays this processing until conflict resolution is complete.

When a non-frozen replica receives a valid $\langle RESOLVE, clientConflictC, w1 \rangle$ request, it proceeds as follows:

- If currentC is later than clientConflictC, or if the viewstamps and timestamps match but the request has already been executed according to the replica's oldOps, the conflict has already been resolved (by contention resolution in the case where the viewstamp in the message is less than vs). The request is handled as a WRITE-1 request.
- Otherwise the replica stores clientConflictC in conflictC and adds w1 to ops if it is not already there. Then it sends a $\langle START, conflictC, ops, currentC, grantTS \rangle_{\sigma_r}$ message to the server proxy code running at the current primary of the BFT protocol.

When the server proxy code running at the primary receives a quorum of valid START messages (including one from itself) it creates a BFT operation to resolve the conflict. The argument to this operation is the quorum of these START messages; call this startQ. Then it causes BFT to operate by passing the operation request to the BFT code running at its node. In other words, the server proxy becomes a client of BFT, invoking an operation on the BFT service implemented by the same replica set that implements the HQ service.

BFT runs in the normal way: the primary orders this operation relative to earlier requests to resolve contention and starts the BFT protocol to obtain agreement on this request and ordering. At the end of agreement each

replica makes an upcall to the server proxy code, passing it startQ, along with the current viewstamp (which has advanced because of the running of BFT).

In response to the upcall, the server proxy code produces the new system state; now we describe this processing. In this discussion we will use the notation startQ.currentC, startQ.ops, etc., to denote the list of corresponding components of the START messages in startQ.

Producing the new state occurs as follows:

- If startQ doesn't contain a quorum of correctly signed START messages, the replica immediately returns from the upcall, without doing any processing. This can happen only if the primary is faulty. The replica makes a call to BFT requesting it to do a view change; when this call returns, it sends its START message to the new primary.
- 2. The replica determines whether startQ.grantTS forms a certificate (i.e., it consists of a quorum of valid matching grants). It chooses the grant certificate if one exists, else the latest valid certificate in startQ.currentC; call this certificate C.
- 3. Next the replica determines whether it needs to undo the most recent operation that it performed prior to conflict resolution; this can happen if some client started phase 2 of a contending write and the replica had executed its WRITE-2 request, yet none of the replicas that contributed to startQ knew about the WRITE-2 and some don't even know of a grant for that request. The replica can recognize the situation because currentC is later than C. To undo, it makes an upcall to the server code, requesting the undo. Then it uses prev to revert the state in oldOps for the client that requested the undone operation, and sets currentC to backupC.
- 4. Next the replica brings itself up to date by executing the operation identified by C, if it hasn't already done so. This processing may include executing even earlier operations, which it can obtain from other replicas if necessary, as discussed in Section 3.4. It executes the operations by making upcalls to the server code and updates its oldOps to reflect the outcome of these executions. Note that all honest replicas will have identical oldOps after this step.
- 5. Next the replica builds an ordered list L of operations that need to be executed. L contains all valid non-duplicate (according to its oldOps) requests in startQ.ops, except that the replica retains at most one operation per client; if a (faulty) client submitted multiple operations (different hashes), it selects one of these in a deterministic way, e.g., smallest

- hash. The operations in L are ordered in some deterministic way, e.g., based on the cid ordering of the clients that requested them.
- 6. The operations in L will be executed in the selected order, but first the replica needs to obtain certificates to support each execution. It updates vs to hold the viewstamp given as an argument of the upcall and sends a grant for each operation at its selected timestamp to all other replicas.
- 7. The replica waits to receive 2f + 1 valid matching grants for each operation and uses them to form certificates. Then it executes the operations in L in the selected order by making upcalls, and updating ops, oldOps, grantTS and currentC (as in Write-2 processing) as these executions occur.
- 8. Finally the replica clears *conflictC* and replies to the RESOLVE request that caused it to freeze (if there is one); this processing is like that of a WRITE-1 request (although most likely a WRITE2-ANS response will be sent).

Conflict resolution has no effect on the processing of WRITE-1 and READ request. However, to process requests that contain certificates (WRITE-2, RESOLVE, and also the write-back requests) the replica must be as up to date as the client with respect to contention resolution. The viewstamp conveys the needed information: if the viewstamp in the certificate in the request is greater than vs, the replica calls down to the BFT code at its node, requesting to get up to date. This call returns the startQ's and viewstamps for all the BFT operations the replica was missing. The replica processes all of this information as described above; then it processes the request as described previously.

A bad primary might not act on START messages, leaving the system in a state where it is unable to make progress. To prevent this, a replica will broadcast the START message to all replicas if it doesn't receive the upcall in some time period; this will cause BFT to do a view-change and switch to a new primary if the primary is the problem. The broadcast is also useful to handle bad clients that send the RESOLVE request to just a few replicas.

3.3.2 Processing at Clients

The only impact of conflict resolution on client processing is that a WRITE-2-ANS response might contain a different certificate than the one sent in the WRITE-2 request; this can happen if contention resolution ran concurrently with the write 2 phase. To handle this case the client selects the latest certificate and uses it to redo the write-2 phase.

3.4 State Transfer

State transfer is required to bring slow replicas up to date so that they may execute more recent writes. A replica detects that it has missed some updates when it receives a valid certificate to execute a write at timestamp t, but has an existing value of currentC.ts smaller than t-1.

A simple but inefficient protocol for state transfer is to request state from all replicas, for each missing update up to t-1, and wait for f+1 matching replies. To avoid transferring the same update from multiple replicas we take an optimistic approach, retrieving a single full copy of updated state, while confirming the hash of the updates from the remaining replicas.

A replica requests state transfer from f+1 replicas, supplying a timestamp interval for the required updates. One replica is designated to return the updates, while f others send a hash over this partial log. Responses are sought from other replicas if the hashes don't agree with the partial log, or after a timeout. Since the partial log is likely to be considerably larger than f hashes, the cost of state transfer is essentially constant with respect to f.

To avoid transferring large partial logs, we propose regular system checkpoints to establish complete state at all replicas [2]. These reduce subsequent writeback cost and allow logs prior to the checkpoint to be discarded. To further minimize the cost of state transfer, the log records may be compressed, exploiting overwrites and application-specific semantics [7]; alternatively, state may be transferred in the form of differences or Merkle trees [15].

3.5 Correctness

This section presents a high-level correctness argument for the HQ protocol. We prove only the safety properties of the system, namely that we ensure that updates in the system are linearizable [6], in that the system behaves like a centralized implementation executing operations atomically one at a time. A discussion of liveness can be found in [4].

To prove linearizability we need to show that there exists a sequential history that looks the same to correct processes as the system history. The sequential history must preserve certain ordering constraints: if an operation precedes another operation in the system history, then the precedence must also hold in the sequential history.

We construct this sequential history by ordering all writes by the timestamp assigned to them, putting each read after the write whose value it returns.

To construct this history, we must ensure that different writes are assigned unique timestamps. The HQ protocol achieves this through its two-phase process — writes

must first retrieve a quorum of grants for the same timestamp to proceed to phase 2, with any two quorums intersecting at at least one non-faulty replica. In the absence of contention, non-faulty replicas do not grant the same timestamp to different updates, nor do they grant multiple timestamps to the same update.

To see preservation of the required ordering constraints, consider the quorum accessed in a READ or WRITE-1 operation. This quorum intersects with the most recently completed write operation at at least one non-faulty replica. At least one member of the quorum must have *currentC* reflecting this previous write, and hence no complete quorum of responses can be formed for a state previous to this operation. Since a read writes back any pending write to a quorum of processes, any subsequent read will return this or a later timestamp.

We must also ensure that our ordering constraints are preserved in the presence of contention, during and following BFT invocations. This is provided by two guarantees:

- Any operation that has received 2f+1 matching WRITE-2-ANS responses prior to the onset of contention resolution is guaranteed to retain its time-stamp t. This follows because at least one non-faulty replica that contributes to startQ will have a currentC such that $currentC.ts \geq t$. Furthermore contention resolution leaves unchanged the order of all operations with timestamps less than or equal to the latest certificate in startQ.
- \bullet No operation ordered subsequently in contention resolution can have a quorum of 2f+1 existing WRITE-2-ANS responses. This follows from the above.

A client may receive up to 2f matching WRITE-2-ANS responses for a given certificate, yet have its operation reordered and committed at a later timestamp. Here it will be unable to complete a quorum of responses to this original timestamp, but rather will see its operation as committed later in the ordering after it redoes its write-2 phase using the later certificate and receives a quorum of WRITE-2-ANS responses.

The argument for safety (and also the argument for liveness given in [4]) does not depend on the behavior of clients. This implies that the HQ protocol tolerates Byzantine-faulty clients, in the sense that they cannot interfere with the correctness of the protocol.

3.6 Transactions

This section describes how we extend our system to support transactions that affect multiple objects.

We extend the client WRITE-1 request so that now it can contain more than one oid; in addition it must provide an op# for each object. Thus we now have $\langle WRITE-1, cid, oid1, op1\#, ..., oidk, opk\#, op\rangle_{\sigma_c}$. We still restrict the client to one outstanding operation per object; this implies that if it performs a multi-object operation, it cannot perform operations on any of the objects used in that operation until it finishes. Note that op could be a sequence of operations, e.g., it consists of $op_1; ...; op_m$, as perceived by the server code at the application.

The requirement for correct execution of a transaction is that for each object it uses it must be executed at the same place in the order for that object at *all* replicas. Furthermore it must not be interleaved with other transactions. For example suppose one transaction consisted of op1(o1); op2(o2) while a second consisted of op3(o1); op4(o2); then the assignment of timestamps cannot be such that o3 happens after o1 while o4 happens before o2.

We achieve these conditions as follows.

- When a replica receives a valid new multi-object request, and it can grant this client the next time-stamp for each object, it returns a multi-grant of the form ⟨cid, h, vs, olist⟩σ_τ, where olist contains an entry ⟨oid, op#, ts⟩ for each object used by the multi-object request; otherwise it refuses, returning all outstanding grants for objects in the request. In either case it returns its most recent certificate for each requested object.
- A client can move to phase 2 if it receives a quorum of matching multi-grants. Otherwise it either does a WRITEBACKWRITE or requests contention resolution. The certificate in the WRITE-2 request contains a quorum of multi-grants; it is valid only if the multi-grants are identical.
- A replica processes a valid WRITE-2 request by making a single upcall to the application. Of course it does this only after getting up to date for each object used by the request. This allows the application to run the transaction atomically, at the right place in the order.
- To carry out a resolve request, a replica freezes for *all* objects in the request and performs conflict resolution for them simultaneously: Its START message contains information for each object identified in the RESOLVE request.
- When processing startQ during contention resolution, a replica retains a most one valid request per client per object. It orders these requests in some deterministic way and sends grants to the other replicas; these will be multi-grants if some of these

request are multi-object operations, and the timestamp for object o will be $o.currentC.ts + |L_o|$, where L_o is L restricted to requests that concern object o. It performs the operations in the selected order as soon as it obtains the newC certificate.

4 Optimizations

There are a number of ways to improve the protocol just described. For example, the WRITE-2-ANS can contain the client op# instead of a certificate; the certificate is needed only if it differs from what the client sent in the request. In addition we don't send certificates in responses to WRITE-1 and READ requests, since these are used only to do writebacks, which aren't needed in the absence of contention and failures; instead, clients need to fetch the certificate from the replicas returning the largest timestamp before doing the writeback. Another useful optimization is to avoid sending multiple copies of results in responses to READ and WRITE-2 requests; instead, one replica sends the answer, while the others send only hashes, and the client accepts the answer if it matches the hashes. Yet another improvement is to provide grants optimistically in responses to WRITE-1 requests: if the replica is processing a valid WRITE-2 it can grant the next timestamp to the WRITE-1 even before this processing completes. (However, it cannot reply to the WRITE-2 until it has performed the operation.)

Below we describe two additional optimizations: early grants and avoiding signing. In addition we discuss preferred quorums, and our changes to BFT.

4.1 Early Grants

The conflict resolution strategy discussed in Section 3.3 requires an extra round of communication at the end of running BFT in order for replicas to obtain grants and build certificates.

We avoid this communication by producing the grants while running BFT. The BFT code at a replica executes an operation by making an upcall to the code running above it (the HQ server code in our system) once it has received a quorum of valid COMMIT messages. We modify these messages so that they now contain grants. This is done by modifying the BFT code so that prior to sending *commit* messages it does a MAKEGRANT upcall to the server proxy code, passing it startQ and the viewstamp that corresponds to the operation being executed. The server code determines the grants it would have sent in processing startQ and returns them in its response; the BFT code then piggybacks the grants on the COMMIT message it sends to the other replicas.

When the BFT code has the quorum of valid COMMIT messages, it passes the grants it received in these mes-

sages to the server proxy code along with startQ and the viewstamp. If none of the replicas that sent COMMIT messages is faulty, the grants will be exactly what is needed to make certificates. If some grants are bad, the replica carries out the post phase as described in Section 3.3.

The grants produced while running BFT could be collected by a malicious intruder or a bad replica. Furthermore, the BFT operation might not complete; this can happen if the BFT replicas carry out a view change, and fewer than f+1 honest replicas had sent out their COMMIT messages prior to the view change. However, the malicious intruder can't make a certificate from grants collected during the running of a single aborted BFT operation, since there can be at most 2f of them, and it is unable to make a certificate from grants produced during the execution of different BFT operations because these grants contain different viewstamps.

4.2 Avoiding Signing

In Section 3, we assumed that grants and WRITE-1 requests were signed. Here we examine what happens when we switch instead to MACs (for WRITE-1 requests) and *authenticators* (for grants). An authenticator [2] is a vector of MACs with an entry for each replica; replicas create authenticators by having a secret key for each other replica and using it to create the MAC for the vector entry that corresponds to that other replica.

Authenticators and MACs work fine if there is no contention and no failures. Otherwise problems arise due to an important difference between signatures and authenticators: A signature that any good client or replica accepts as valid will be accepted as valid by all good clients and replicas; authenticators don't have this property. For example, when processing startQ replicas determined the most recent valid certificate; because we assumed signatures, we could be sure that all honest replicas would make the same determination. Without signatures this won't be true, and therefore we need to handle things differently.

The only place where authenticators cause problems during non-contention processing is in the responses to WRITE-2 and writeback requests. In the approach described in Section 3.2, replicas drop bad WRITE-2 and writeback requests. This was reasonable when using signatures, since clients can avoid sending bad certificates. But clients are unable to tell whether authenticators are valid; they must rely on replicas to tell them.

Therefore we provide an additional response to WRITE-2 and writeback requests: the replica can send a WRITE-2-REFUSED response, containing a copy of the certificate, and signed by it. When a client receives such a response it requests contention resolution.

The main issue in contention resolution is determining the latest valid certificate in startQ. It doesn't work to just select the certificate with the largest timestamp, since it might be forged. Furthermore there might be two or more certificates for the same highest timestamp but different requests; the replicas need to determine which one is valid.

We solve these problems by doing some extra processing before running BFT. Here is a sketch of how it works:

To solve the problem of conflicting certificates that propose the same timestamp but for different requests, the primary builds startQ from START messages as before except that startQ may contain more than a quorum of messages. The primary collects START messages until there is a subset $startQ_{sub}$ that contains no conflicting certificates. If two START messages propose conflicting certificates, neither is placed in $startQ_{sub}$; instead the primary adds another message to startQ and repeats the analysis. It is safe for the primary to wait for an additional message because at least one of the conflicting messages came from a dishonest replica.

This step ensures that $startQ_{sub}$ contains at most one certificate per timestamp. It also guarantees that at least one certificate in $startQ_{sub}$ contains a timestamp greater than or equal to that of the most recently committed write operation because $startQ_{sub}$ contains at least f+1 entries from non-faulty replicas, and therefore at least one of them supplies a late enough certificate.

The next step determines the latest valid certificate. This is accomplished by a voting phase in which replicas collect signed votes for certificates that are valid for them and send this information to the primary in signed ACCEPT messages; the details can be found in [4]. The primary collects a quorum of ACCEPT messages and includes these messages as an extra argument in the call to BFT to execute the operation. Voting can be avoided if the latest certificate was formed from startQ.grantTS or proposed by at least f+1 replicas.

This step retains valid certificates but discards forged certificates. Intuitively it works because replicas can only get votes for valid certificates.

When replicas process the upcall from BFT, they use the extra information to identify the latest certificate. An additional point is that when replicas create the set L of additional operations to be executed, they add an operation to L only if it appears at least f+1 times in startQ.ops. This test ensures that the operation is vouched for by at least one non-faulty replica, and thus avoids executing forged operations.

This scheme executes fewer requests than the approach discussed in Section 3.3. In particular, a write request that has already reached phase 2 will be executed in the scheme discussed in Section 3.3, but now it might not be (because it doesn't appear at least f+1 times in

startQ.ops). In this case when the WRITE-2 request is processed by a replica after contention resolution completes, the replica cannot honor the request. Instead it sends a WRITE-2-RETRY response containing a grant for the next timestamp, either for this client or some other client. When a client gets this response, it re-runs phase 1 to obtain a new certificate before retrying phase 2.

4.3 Preferred Quorums

With preferred quorums, only a predetermined quorum of replicas carries out the protocol during fault-free periods. This technique is used in Q/U and is similar to the use of witnesses in Harp [10]. In addition to reducing cost, preferred quorums ensure that all client operations intersect at the same replicas, reducing the frequency of writebacks.

Since ultimately every replica must perform each operation, we have clients send the WRITE-1 request to all replicas. However, only replicas in the preferred quorum respond, the authenticators in these responses contain entries only for replicas in the preferred quorum, and only replicas in the preferred quorum participate in phase 2. If clients are unable to collect a quorum of responses, they switch to an unoptimized protocol using a larger group.

Replicas not in the preferred quorum need to periodically learn the current system state, in particular the timestamp of the most recently committed operation. This communication can be very lightweight, since only metadata and not client operations need be fetched.

4.4 BFT Improvements

The original implementation of BFT was optimized to perform well at small f, e.g., at f=2. Our implementation is intended to scale as f increases. One main difference is that we use TCP instead of UDP, to avoid costly message loss in case of congestion at high f. The other is the use of MACs instead of authenticators in protocol messages. The original BFT used authenticators to allow the same message to be broadcast to all other replicas with a single operating system call, utilizing IP multicast if available. However, authenticators add linearly-scaling overhead to each message, with this extra cost becoming significant at high f in a non-broadcast medium.

Additionally, our implementation of BFT allows the use of preferred quorums.

5 Analysis

Here we examine the performance characteristics of HQ, BFT, and Q/U analytically; experimental results can be found in Section 6. We focus on the cost of write operations since all three protocols offer one-phase read opera-

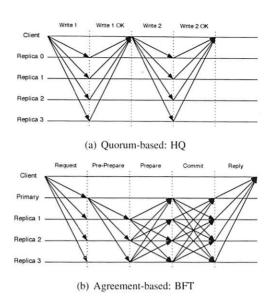


Figure 2: Protocol communication patterns.

tions, and we expect similar performance in this case. We also focus on performance in the normal case of no failures and no contention. For both HQ and Q/U we use assume preferred quorums and MACs/authenticators. We show results for the original BFT algorithm (using authenticators and without preferred quorums), BFT-MACs (using MACs but not preferred quorums), and BFT-opt (using both MACs and preferred quorums). We assume the protocols use point-to-point communication.

Figure 2 shows the communication patterns for BFT and HQ; the communication pattern for Q/U is similar to the first round of HQ, with a larger number of replicas. Assuming that latency is dominated by the number of message delays needed to process a request, we can see that the latency of HQ is lower than that of BFT and the latency for Q/U is half of that for HQ. One point to note is that BFT can be optimized so that replicas reply to the client following the *prepare* phase, eliminating *commit*-phase latency in the absence of failures; with this optimization BFT can achieve the same latency as HQ. However, to amortize its quadratic communication costs, BFT employs batching, committing a group of operations as a single unit. This can lead to additional latency over a quorum-based scheme.

Figure 3 shows the total number of messages required to carry out a write request in the three systems; the figure shows the load at both clients and servers. Consider first Figure 3a, which shows the load at servers. In both HQ and Q/U, servers process a constant number of messages to carry out a write request: 4 messages in HQ and 2 in Q/U. In BFT, however, the number of messages is linear in f: For each write operation that runs through

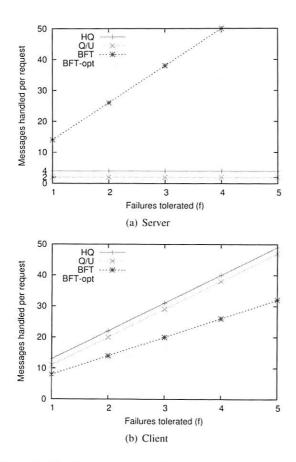


Figure 3: Total messages sent/received per write operation in BFT, Q/U, and HQ

BFT, each replica must process 12f+2 messages. This is reduced to 8f+2 messages in BFT-opt by using preferred quorums.

Figure 3b shows the load at the client. Here we see that BFT-opt has the lowest cost, since a client just sends the request to the replicas and receives a quorum of responses. Q/U also requires one message exchange, but it has larger quorums (of size 4f+1), for 9f+2 messages. HQ has two message exchanges but uses quorums of size 2f+1; therefore the number of messages processed at the client, 9f+4, is similar in HQ to Q/U.

Figure 4 shows the total byte count of the messages processed to carry out a write request. This is computed using 20 byte SHA-1 digests [17] and HMAC authentication codes [16], 44 byte TCP/IP overhead, and a nominal request payload of 256 bytes. We analyze the fully optimized version of Q/U, using *compact timestamps* and *replica histories* pruned to the minimum number of two *candidates*. The results for BFT in Figure 4a show that our optimizations (MACs and preferred quorums) have a major impact on the byte count at replicas. The use of MACs causes the number of bytes to grow only linearly

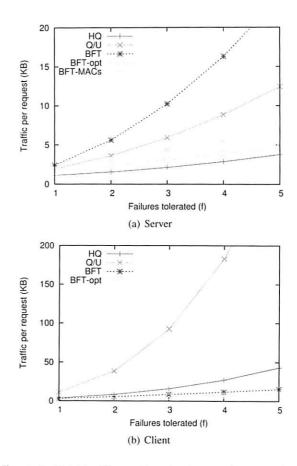


Figure 4: Total traffic sent/received per write operation in BFT, Q/U, and HQ

with f as opposed to quadratically as in BFT, as shown by the BFT-MACs line; an additional linear reduction in traffic occurs through the use of preferred quorums, as shown by BFT-opt line.

Figure 4a also shows results for HQ and Q/U. In HQ the responses to the WRITE-1 request contains an authenticator and the WRITE-2 request contains a certificate, which grows quadratically with f. Q/U is similar: The response to a write returns what is effectively a grant (replica history), and these are combined to form a certificate (object history set), which is sent in the next write request. However, the grants in Q/U are considerably larger than those in HQ and also contain bigger authenticators (size 4f + 1 instead of 2f + 1), resulting in more bytes per request in Q/U than HQ. While HQ and Q/U are both affected by quadratically-sized certificates, this becomes a problem more slowly in HQ: At a given value of f = x in Q/U, each certificate contains the same number of grants as in HQ at f = 2x.

Figure 4b shows the bytes required at the client. Here the load for BFT is low, since the client simply sends the request to all replicas and receives the response. The load

for Q/U is the highest, owing to the quadratically growing certificates, larger grants and communication with approximately twice as many replicas.

6 Experimental Evaluation

This section provides performance results for HQ and BFT in the case of no failures. Following [1], we focus on performance of writes in a counter service supporting increment and fetch operations. The system supports multiple counter objects; each client request involves a single object and the client waits for a write to return before executing the subsequent request. In the noncontention experiments different clients use different objects; in the contention experiments a certain percentage of requests goes to a single shared object.

To allow meaningful comparisons of HQ and BFT, we produced new implementations of both, derived from a common C++ codebase. Communication is implemented over TCP/IP sockets, and we use SHA-1 digests for HMAC message authentication. HQ uses preferred quorums; BFT-MACs and BFT-opt use MACs instead of authenticators, with the latter running preferred quorums. Client operations in the counter service consist of a 10 byte *op* payload, with no disk access required in executing each operation.

Our experiments ran on Emulab [20], utilizing 66 pc3000 machines. These contain 3.0 GHz 64-bit Xeon processors with 2GBs of RAM, each equipped with gigabit NICs. The emulated topology consists of a 100Mbps switched LAN with near-zero latency, hosted on a gigabit backplane with a Cisco 6509 high-speed switch. Network bandwidth was not found to be a limiting factor in any of our experiments. Fedora Core 4 is installed on all machines, running Linux kernel 2.6.12.

Sixteen machines host a single replica each, providing support up to f=5, with each of the remaining 50 machines hosting two clients. We vary the number of logical clients between 20 and 100 in each experiment, to obtain maximum possible throughput. We need a large number of clients to fully load the system because we limit clients to only one operation at a time.

Each experiment runs for 100,000 client operations of burn-in time to allow performance to stabilize, before recording data for the following 100,000 operations. Five repeat runs were recorded for each data-point, with the variance too small to be visible in our plots. We report throughput; we observed batch size and protocol message count in our experiments and these results match closely to the analysis in Section 5.

We begin by evaluating performance when there is no contention: we examine maximum throughput in HQ and BFT, as well as their scalability as f grows. Throughput

is CPU-bound in all experiments, hence this figure reflects message processing expense and cryptographic operations, along with kernel message handling overhead.

Figure 5 shows that the lower message count and fewer communication phases in HQ is reflected in higher throughput. The figure also shows significant benefits for the two BFT optimizations; the reduction in message size achieved by BFT-MACs, and the reduced communication and cryptographic processing costs in BFT-opt.

Throughput in HQ drops by 50% as f grows from 1 to 5, a consequence of the overhead of computing larger authenticators in grants, along with receiving and validating larger certificates. The BFT variants show slightly worse scalability, due to the quadratic number of protocol messages.

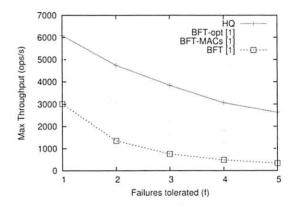


Figure 5: Maximum non-batched write throughput under varying f.

Based on our analysis, we expect Q/U to provide somewhat less than twice the throughput of HQ at f=1, since it requires half the server message exchanges but more processing per message owing to larger messages and more MAC computations. We also expect it to scale less well than HQ, since its messages and processing grow more quickly with f than HQ's.

The results in Figure 5 don't tell the whole story. BFT can batch requests: the primary collects messages up to some bound, and then runs the protocol once per batch. Figure 6 shows that batching greatly improves BFT performance. The figure shows results for maximum batch sizes of 2, 5, and 10; in each case client requests may accumulate for up to 5ms at the primary, yielding observed batch sizes very close to the maximum.

Figure 7 shows the performance of HQ for f=1,...,5 in the presence of write contention; in the figure *contention factor* is the fraction of writes executed on a single shared object. The figure shows that HQ performance degrades gracefully as contention increases. Performance reduction flattens significantly for high rates

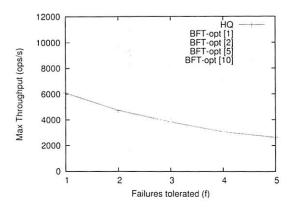


Figure 6: Effect of BFT batching on maximum write throughput.

of write contention because multiple contending operations are ordered with a single round of BFT, achieving a degree of write batching. For example, at f=2 this batching increases from an average of 3 operations ordered per round at contention factor 0.1 to 16 operations at contention factor 1.

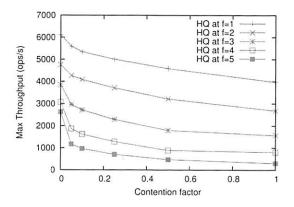


Figure 7: HQ throughput under increasing write contention.

7 Related Work

Byzantine quorum systems were introduced by Malkhi and Reiter [12]. The initial constructions were designed to handle only read and (blind) write operations. These were less powerful than state machine replication (implemented by our protocols) where the outcome of an operation can depend on the history of previously executed operations. Byzantine quorum protocols were later extended to support general object replication assuming benign clients (e.g. [13, 3]), and subsequently to support Byzantine clients but for larger non-blocking quo-

rums [5]. Still more recent work showed how to handle Byzantine clients while using only 3f + 1 replicas [11].

Recently, Abd-El-Malek et al. [1] demonstrated for the first time how to adapt a quorum protocol to implement state machine replication for multi-operation transactions with Byzantine clients. This is achieved with a combination of optimistic versioning and by having client requests store a history of previous operations they depend on, allowing the detection of conflicts in the ordering of operations (due to concurrency or slow replicas) and the retention of the correct version.

Our proposal builds on this work but reduces the number of replicas from 5f + 1 to 3f + 1. Our protocol does not require as many replicas owing to our mechanism for detecting and recovering from conflicting orderings of concurrent operations at different replicas. The Q/U protocols use a one-phase algorithm for writes; Abd-El-Malek et al show in their paper that their one-phase write protocol cannot run with fewer than 5f + 1 replicas (with quorums of size 4f + 1). We use a two-phase write protocol, allowing us to require only 3f + 1 replicas. A further difference of our work from Q/U is our use of BFT to order contending writes; this hybrid approach resolves contention much more efficiently than the approach used in Q/U, which resorts to an exponential backoff of concurrent writers that may lead to a substantial performance degradation.

In work done concurrently with that on Q/U, Martin and Alvisi [14] discuss the tradeoff between number of rounds and number of replicas for reaching *agreement*, a building block that can be used to construct state machine replication. They prove that 5f+1 replicas are needed to ensure reaching agreement in two communication steps and they present a replica-based algorithm that shows this lower bound is tight.

Earlier proposals for Byzantine fault tolerant statemachine replication (e.g., Rampart [18] and BFT [2]) relied on inter-replica communication, instead of client-controlled, quorum-based protocols, to serialize requests. These protocols employ 3f+1 replicas, and have quadratic communication costs in the normal case, since each operation involves a series of rounds where each replica sends a message to all remaining replicas, stating their agreement on an ordering that was proposed by a primary replica. An important optimization decouples the agreement from request execution [21] reducing the number of the more expensive storage replicas to 2f+1 but still retaining the quadratic communication costs.

8 Conclusions

This paper presents HQ, a new protocol for Byzantinefault-tolerant state-machine replication. HQ is a quorum based protocol that is able to run arbitrary operations. It reduces the required number of replicas from the 5f+1 needed in earlier work (Q/U) to the minimum of 3f+1 by using a two-phase instead of a one-phase write protocol.

Additionally we present a new way of handling contention in quorum-based protocols: we use BFT. Thus we propose a hybrid approach in which operations normally run optimistically, but a pessimistic approach is used when there is contention. The hybrid approach can be used broadly; for example it could be used in Q/U to handle contention, where BFT would only need to run at a predetermined subset of 3f+1 replicas.

We also presented a new implementation of BFT that was developed to scale with f.

Based on our analytic and performance results, we believe the following points are valid:

- In the region we studied (up to f = 5), if contention is low and low latency is the main issue, then if it is acceptable to use 5f + 1 replicas, Q/U is the best choice else HQ is best since it outperforms BFT with a batch size of 1.
- Otherwise, BFT is the best choice in this region: it can handle high contention workloads, and it can beat the throughput of both HQ and Q/U through its use of batching.
- Outside of this region, we expect HQ will scale best.
 Our results show that as f grows, HQ's throughput decreases more slowly than Q/U's (because of the latter's larger messages and processing costs) and BFT's (where eventually batching cannot compensate for the quadratic number of messages).

9 Acknowledgments

We thank the anonymous reviewers and our shepherd Mema Roussopoulos for their valuable feedback and the developers of Q/U for their cooperation. We also thank the supporters of Emulab, which was absolutely crucial to our ability to run experiments.

This research was supported by NSF ITR grant CNS-0428107 and by T-Party, a joint program between MIT and Quanta Computer Inc., Taiwan.

References

- [1] ABD-EL-MALEK, M., GANGER, G. R., GOODSON, G. R., REITER, M. K., AND WYLIE, J. J. Fault-scalable byzantine fault-tolerant services. In SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles (New York, NY, USA, 2005), ACM Press, pp. 59–74.
- [2] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance and Proactive Recovery. ACM Transactions on Computer Systems 20, 4 (Nov. 2002), 398–461.

- [3] CHOCKLER, G., MALKHI, D., AND REITER, M. Backoff protocols for distributed mutual exclusion and ordering. In Proc. of the IEEE International Conference on Distributed Computing Systems (2001).
- [4] COWLING, J., MYERS, D., LISKOV, B., RODRIGUES, R., AND SHRIRA, L. Hq replication: Properties and optimizations. Technical Memo In Prep., MIT Computer Science and Artificial Laboratory, Cambridge, Massachusetts, 2006.
- [5] FRY, C., AND REITER, M. Nested objects in a byzantine Quorum-replicated System. In Proc. of the IEEE Symposium on Reliable Distributed Systems (2004).
- [6] HERLIHY, M. P., AND WING, J. M. Axioms for Concurrent Objects. In Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages (1987).
- [7] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected Operation in the Coda File System. In *Thirteenth ACM Symposium on Operating Systems Principles* (Asilomar Conference Center, Pacific Grove, CA., Oct. 1991), pp. 213–225.
- [8] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems 4, 3 (July 1982), 382–401.
- [9] LAMPORT, L. L. The implementation of reliable distributed multiprocess systems. Computer Networks 2 (1978), 95–114.
- [10] LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. Replication in the Harp File System. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles* (Pacific Grove, California, 1991), pp. 226–238.
- [11] LISKOV, B., AND RODRIGUES, R. Byzantine clients rendered harmless. Tech. Rep. MIT-LCS-TR-994 and INESC-ID TR-10-2005, July 2005.
- [12] MALKHI, D., AND REITER, M. Byzantine Quorum Systems. Journal of Distributed Computing 11, 4 (1998), 203–213.
- [13] MALKHI, D., AND REITER, M. An Architecture for Survivable Coordination in Large Distributed Systems. *IEEE Transactions* on Knowledge and Data Engineering 12, 2 (Apr. 2000), 187–202.
- [14] MARTIN, J.-P., AND ALVISI, L. Fast byzantine consensus. In International Conference on Dependable Systems and Networks (2005), IEEE, pp. 402–411.
- [15] MERKLE, R. C. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - Crypto*'87, C. Pomerance, Ed., no. 293 in Lecture Notes in Computer Science. Springer-Verlag, 1987, pp. 369–378.
- [16] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Fips 198: The keyed-hash message authentication code (hmac), March 2002.
- [17] NATIONAL INSTITUTE OF STANDARDS AND TECNOLOGY. Fips 180-2: Secure hash standard, August 2002.
- [18] REITER, M. The Rampart toolkit for building high-integrity services. Theory and Practice in Distributed Systems (Lecture Notes in Computer Science 938) (1995), 99–110.
- [19] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22, 4 (1990), 299–319.
- [20] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GU-RUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), USENIX Association, pp. 255–270.
- [21] YIN, J., MARTIN, J., VENKATARAMANI, A., ALVISI, L., AND DAHLIN, M. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the 19th ACM Sympo*sium on Operating Systems Principles (Oct. 2003).

BAR Gossip

Harry C. Li, Allen Clement, Edmund L. Wong, Jeff Napper, Indrajit Roy,
Lorenzo Alvisi, Michael Dahlin

Laboratory for Advanced Systems Research (LASR),

Dept. of Computer Sciences,

The University of Texas at Austin

We present the first peer-to-peer data streaming application that guarantees predictable throughput and low latency in the BAR (Byzantine/Altruistic/Rational) model, in which nonaltruistic nodes can behave in ways that are self-serving (rational) or arbitrarily malicious (Byzantine). At the core of our solution is a BARtolerant version of gossip, a well-known technique for scalable and reliable data dissemination. BAR Gossip relies on verifiable pseudo-random partner selection to eliminate non-determinism that can be used to game the system while maintaining the robustness and rapid convergence of traditional gossip. A novel fair enough exchange primitive entices cooperation among selfish nodes on short timescales, avoiding the need for long-term node reputations. Our initial experience provides evidence for BAR Gossip's robustness. Our BAR-tolerant streaming application provides over 99% convergence for broadcast updates when all clients are selfish but not colluding, and over 95% convergence when up to 40% of clients collude while the rest follow the BAR Gossip also performs well when protocol. the client population consists of both selfish and Byzantine nodes, achieving over 93% convergence even when 20% of the nodes are Byzantine.

1 Introduction

Streaming media is an increasingly useful application at several scales of deployment. For example, the 2006 NCAA tournament had peak participation of 150k+ users for their live streaming services. At smaller scales, such as academic conferences like OSDI or artistic events like Austin's SXSW that draw audiences of dozens to hundreds of people, there is interest in providing a live stream, but little existing infrastructure to support it.

At all these scales, a peer-to-peer (p2p) streaming solution appears to be an intriguing alternative to tradi-

tional methods. One advantage of p2p systems is their potential to be highly robust, scalable, and adaptive. For example, a p2p architecture could, in principle, absorb the impact of an unexpected flash crowd. Furthermore, large-scale content providers may adopt p2p-based solutions to shift costs (like bandwidth) to clients, and small-scale providers might find it simpler to use a self-organizing p2p network instead of provisioning and maintaining a large dedicated server.

Realizing the promises of p2p in streaming services is non-trivial. First, the service must guarantee highly reliable, stable, and timely throughput of messages despite the presence of faulty, misconfigured, or even malicious peers. Second, the service must be robust against selfish users, who try to catch a free ride by receiving streams without contributing their fair share to other users [12]. The free-rider phenomenon in p2p systems is a symptom of a broader issue: any system deployed across multiple administrative domains must be designed for the possibility that nodes will deviate from their specification if doing so is advantageous. File-sharing p2p applications, like BitTorrent [7], have recognized this issue and introduced a set of heuristics to incentivize faithful participation—these heuristics, however, optimize for bulk file transfer, not for the timely delivery of a series of small frames required by a streaming service.

This paper presents the first p2p streaming media application designed for a system model (BAR [1]) in which *altruistic* nodes (who follow the protocol assigned to them) coexist with both arbitrarily malicious (*Byzantine*) and self-serving (*rational*) nodes. Our BARtolerant solution, based on gossip protocols [4, 8, 19, 35], provides a scalable mechanism for information dissemination that ensures predictable throughput even if all of the nodes act selfishly and the remainder act maliciously or malfunction in arbitrary ways.

The defining characteristic of gossip protocols is that each node exchanges data, or gossips, with randomly selected peers: it is precisely this randomness that gives gossip protocols their enviable robustness. From the perspective of designing BAR-tolerant protocols, however, randomness can be a real headache: in fact, *any* source of non-determinism is hard to deal with in the BAR model because it gives opportunities for rational users to hide selfish actions in the guise of legitimate, non-deterministic behavior.

We overcome this difficulty by building a BARtolerant gossip protocol that uses verifiable pseudorandomness as the means for peer selection: in particular, we exploit properties of pseudo-random number generators and unique signature schemes to build a verifiable pseudo-random partner selection algorithm. This approach eliminates the main source of non-determinism in traditional gossip-randomness in partner selectionyet maintains the unpredictability and rapid convergence of traditional gossip. Our novel peer selection technique, in combination with a simple fair enough exchange mechanism based on the notion of credible threats [1, 9], proves effective in encouraging the fair exchange [17] of one node's updates for another's. We believe that enticing cooperation over short timescales is simpler and more robust than approaches based on long-term reputations [16, 23, 36] because doing so limits Brutus attacks, in which malicious nodes maximize damage by earning the trust of their victims before striking.

We build a prototype streaming application that uses our BAR Gossip protocol to provide a stable throughput multicast. We show that BAR Gossip is robust to Byzantine and selfish behavior, even when 40% of selfish nodes collude. In an environment in which 20% of clients are Byzantine and the remaining are rational, we demonstrate that if the broadcaster can multicast packets to a constant number of random nodes, then non-Byzantine clients can reliably deliver the stream in a timely manner.

In this paper, we make two main technical contributions:

- We design BAR Gossip, the first gossip protocol resilient to both Byzantine and selfish behavior.
- We use BAR gossip to build the first p2p streaming application to guarantee predictable throughput and low latency in the BAR model.

More broadly, by showing that it is possible to derive BAR tolerant versions of a highly non-deterministic protocol such as gossip, we believe this work strengthens the case for BAR tolerance as the right model for reasoning about the dependability of systems deployed across multiple administrative domains.

We introduce the paper in Section 1, which ends with this paragraph outlining the rest of the paper. Section 2 frames our contributions in the context of previous work. The system model is described in Section 3. Section 4 then describes BAR Gossip, followed by Section 5, which supports our claim that BAR Gossip is a robust streaming protocol through a combination of simulations and live experiments.

2 Related Work

BAR Gossip targets streaming of live content among Byzantine, altruistic, and rational nodes. It draws on a broad literature of bulk file transfer systems designed to tolerate node misbehavior as well as a large number of efforts to use gossip for robust data dissemination.

Bulk file transfer. Several systems have addressed selfish behavior in p2p content distribution. BitTorrent [7] leverages a local reputation scheme for filesharing in which nodes give preference to those peers who have reliably reciprocated in the past.

Scrivener [27] generalizes BitTorrent by supporting a distributed reputation scheme based on credits that can be earned and redeemed across multiple files: through this mechanism, a Scrivener node that has been a good citizen can enlist the help of its peers even if the file it wants to acquire is unpopular.

Habib and Chuang [15] study p2p streaming of nonlive media in a selfish environment. They use distributed reputations in the form of global rankings to deter freeriders while providing good quality streams to cooperative clients.

FOX [18] guarantees optimal download time to all the nodes interested in acquiring the same file under the assumption that all nodes are selfish. This guarantee, however, comes at the expense of robustness: the system's incentive structure depends on the fear of mutual assured destruction, and a single Byzantine node can cause the entire system to collapse.

Splitstream [6] is a tree-based multicast protocol that achieves load balancing by dividing content into multiple stripes, each of which is multicast using a separate tree. Splitstream is vulnerable to freeloaders. Ngan et al. [30] observe that if Splitstream's multicast trees are periodically rebuilt and nodes maintain local reputations regarding nodes that have misbehaved, then upstream nodes in a given tree have an incentive to provide good service to downstream nodes to avoid future retaliation.

BAR Gossip differs from these systems in four key ways. First, these systems work to optimize average download bandwidth over long periods of time and do not attempt to maintain stable throughput over shorter intervals. In contrast, our protocol is designed to disseminate live streams and therefore values the highly reliable, stable, and timely throughput that comes with gossipstyle data dissemination. Second, several of these systems are designed to be robust to Byzantine [6] or ratio-

nal [15, 18, 27] players but not both. Third, all of these systems transfer a large collection of file blocks. In contrast, BAR Gossip distributes live streams and must cope with having a relatively small window of "useful" data in flight at any given time; ensuring timely delivery of a small set of data is one of the key challenges in our protocol. Fourth, most of these systems make use of local [7, 30] or distributed [15, 27] node reputations in their incentive structure. But given our desire to provide stable throughput over short periods of time, relying on a node's long-term reputation to predict its short-term behavior is problematic. Furthermore, because gossip partners are likely to change in every round, in our protocol it is virtually impossible for a node to build enough good will with specific partners to support a purely local reputation scheme à la BitTorrent. Additionally, it appears challenging to implement a strategy-proof reputation system in an environment with both rational and Byzantine players.

Gossip. Gossip algorithms were first introduced by Demers et al. [8] to manage replica consistency in the Xerox Clearinghouse Service [31]. Following Birman et al.'s highly influential paper on Bimodal Multicast [4], gossip algorithms have established themselves as one of the leading approaches to achieve reliable and scalable application-level multicast [5, 11, 14, 19, 25, 40]. Gossip protocols are also at the core of a new generation of scalable distributed protocols for failure detection [35], group communication [13, 39], and the monitoring and management of large distributed systems [38].

Experience with the CoolStream implementation of the DONet p2p overlay network [42] makes a strong case for the scalability of gossip-based dissemination of live streams and for its potential to deliver high quality endto-end user experience in the presence of altruistic nodes.

Secure gossiping aims to prevent Byzantine nodes from spreading false updates. While digital signatures are not necessary to accomplish this goal [20–22, 26], they considerably simplify protocol design and are assumed in most practical gossip-based systems [5, 38, 39], including BAR Gossip.

DRUM [2] assumes secure gossip and focuses on fighting denial of service (DoS) attacks through two main techniques: bounding the resources allocated to each gossip operation and directing these operations to random ports.

BAR replication. Aiyer et al. [1] define a replicated state machine protocol under the BAR model. Although the replication overheads of that approach are too high for live streaming media, we draw on many of the same design principles: *predictable communication patterns* manifest in our partner selection algorithm (Section 4.2.1), *cost balancing* manifests in our optimistic

push algorithm (4.2.5), *credible threats* manifest in our key exchange (4.2.4), and *ensuring long term benefit* manifests in our delayed gratification protocol structure (4.1).

3 Model

We consider the problem of streaming a live event across the Internet where the audience members, which we call *clients*, help disseminate stream packets.

Clients can be Byzantine, altruistic, or rational. Each rational client follows a strategy that maximizes its utility. Rational clients share a utility function that describes costs and benefits; in this paper, the benefit consists in the ability to play the live stream and the costs are incurred by sending and receiving packets. A rational client deviates from the specification if and only if doing so increases that client's expected utility. We assume that a rational client's benefit from timely acquisition of each stream packet significantly exceeds the communication costs incurred in doing so.

We assume rational clients ignore messages outside of the protocol. We root this assumption in the existing game theory literature, which considers extra-protocol messages to be outside the strategy space. Excluding these messages is reasonable because convincing clients to act on messages outside the protocol specified by the broadcaster requires i) convincing clients to install the new protocol in addition to BAR Gossip and ii) ensuring that rational clients expect the protocol to provide current and future updates in a BAR environment without support from the live event's source. If this assumption is violated - e.g. alternative, cheaper, better source of the update appears - then liveness guarantees may be eroded. With this assumption, we can demonstrate that, absent other avenues for receiving the desired updates, rational clients benefit from running our protocol.

Altruistic clients follow the protocol as given regardless of costs, similar to correct clients in the traditional fault-tolerance literature. Byzantine clients behave arbitrarily or according to some unspecified utility function.

Non-Byzantine clients maintain clocks synchronized within δ seconds of each other and communicate over point-to-point, synchronous, and unreliable links using both TCP and UDP. When messages are exchanged using UDP, a node that does not receive an expected message assumes that the link dropped it.

We assume that clients subscribe to the live broadcast prior to its start and that non-Byzantine clients remain in the system for the duration of the broadcast. Future work is needed to extend the protocol to accommodate dynamic membership. Given our protocol's reliance on short-term incentives rather than long-term reputations, we are optimistic that such extensions will be feasible. We also assume that each participant is limited to one identity. To mitigate the threat of Sybil attacks [10], it must be hard for participants to collect multiple identities. There exist sophisticated techniques to combat Sybil attacks [37,41]. In our prototype, we take the simple approach of limiting each IP address to one identity.

We make the standard assumption that cryptographic primitives, like digital signatures, symmetric key encryption, and one-way functions, cannot be subverted. Our protocol also requires that each private key generates unique signatures: for a given m, there must exist exactly one valid signature of m by i. Although a number of standard signature algorithms fail to provide this property [29], there exist algorithms that do [3]. We denote a message m signed by i as $\langle m \rangle_i$.

We hold clients accountable for the messages they sign. We define a proof of misbehavior (POM) to be a sequence of signed messages that proves a client sent a message inconsistent with the protocol specification. A POM against a client is sufficient evidence to *evict* that client from the population. Assuming that rational clients gain benefit from being members of the population, we model the system as an infinite game or one with an unpredictable end time so that rational clients are cautious and do not risk eviction by sending POMs. Additional work is needed to determine if there are significant endgame effects for known duration events.

4 BAR Gossip Design

The BAR Gossip protocol describes a method for an altruistic *broadcaster* to stream a live event to a pool of clients. Streaming a live event requires that BAR Gossip guarantee two properties: *i*) non-Byzantine clients do not deliver unauthentic stream packets (i.e., packets not generated by the broadcaster) and *ii*) every altruistic client receives a large fraction of all stream packets in a timely manner. As we later show, although we can provide the first property in all situations, the second property is elusive, and we can only provide it under good network conditions and with a limited number of Byzantine clients.

Before the start time, each client generates a session key pair consisting of a public and private key. Clients sign up for the event by divulging both keys to the broadcaster. The broadcaster then verifies the keys, closes the sign up service, and posts a list that contains each client's identity, address, and public key. Clients sign protocol messages using their private keys to provide authentication, integrity, and non-repudiation of message contents.

During the live event, the broadcaster divides the stream into discrete fixed-size chunks that we call *updates*. We structure BAR Gossip as a sequence of rounds of duration $T + \delta$ where updates are sent by the broadcaster and exchanged among clients. T is a time interval sufficient to complete the per-round message exchanges

required by our protocol. Round zero begins when the live stream starts. Each update expires deadline rounds after it was sent by the broadcaster. When an update expires, all clients that possess that update deliver it to their media players. We consider an update delivered by its deadline to be timely.

In each round r, the broadcaster multicasts ups_per_round updates to subsets of clients. Specifically, for each update, the broadcaster selects nSeeds random clients to receive the update, signs the update, and multicasts it to the selected clients. We require nSeeds to be large enough to guarantee that with high probability at least one receiver is non-Byzantine.

A client is unlikely to receive all updates directly from the broadcaster and relies on two protocols to garner the remaining: Balanced Exchange and Optimistic Push. The Balanced Exchange Protocol allows clients to trade updates one-for-one. That is, if client S has ten updates to offer client R and R has only five to offer in return, then S and R trade five updates in each direction. Each balanced exchange is *incentive-compatible* [33]—rational clients are motivated to follow its steps faithfully because no unilateral deviation from the specified protocol can increase a client's expected utility.

Using the Balanced Exchange Protocol alone, however, is insufficient if a client falls behind in obtaining updates (through bad luck or transient network failures) because that client has little to offer in exchanges. Once behind, a client may continue to fall farther behind. The Optimistic Push Protocol provides a safety net. We call this protocol optimistic because an initiator S is willing to forward useful updates to R in the hope, rather than the certainty, that R will return the favor. In this case, an unequal number of updates may be exchanged—if R has fallen behind, S helps R even if R cannot fully reciprocate.

Although Optimistic Push is not an incentive-compatible protocol, we structure it to encourage rational clients' participation, and our experimental evidence suggests that a rational client will often benefit from active participation in Optimistic Push. In the next subsections, we detail both protocols, prove Balanced Exchange's incentive-compatibility and discuss rational deviations within Optimistic Push. For reference, Figure 1 illustrates both Balanced Exchange and Optimistic Push.

4.1 Balanced Exchange

Balanced Exchange provides an incentive-compatible mechanism for rational clients to exchange updates. In balanced exchanges, each party determines the largest number of new updates it can exchange while keeping the trade equal. A client concurrently executes two tasks: *i*) initiating an exchange with another client and *ii*) responding to Balanced Exchange requests from other

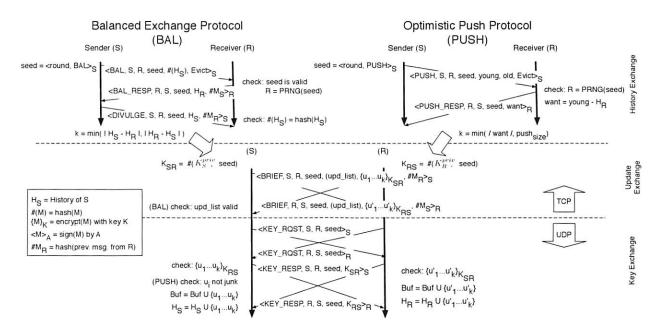


Figure 1: Balanced Exchange and Optimistic Push Protocols for node A contacting node B. During the update exchange, the updalist is sent in the Balanced Exchange protocol, but omitted for Optimistic Push.

clients. As Figure 1 details, an exchange consists of four phases. In the first, partner selection, a client selects another client with whom to trade. In the second, history exchange, the two parties learn about the unexpired updates the other party holds and determines the largest number k of updates that can be exchanged on a one-for-one basis. In the third phase, update exchange, each party deterministically generates an encryption key based upon its private key and a per-exchange seed value. Each party then encrypts its k most recent exchangeable updates and copies the encrypted updates into a briefcase that is sent to the other party. In the fourth phase, key exchange, the parties swap keys and decrypt the contents of received briefcases. A client ends an exchange early in the history exchange phase if that client realizes the exchange will ultimately trade no updates. An exchange completes if both clients execute all four phases or if one of them ends the exchange early as allowed by the protocol. Clients communicate using TCP in the first three phases, and switch to UDP in the fourth: we discuss the reason for this choice in Section 4.2.4, where we consider the incentives that motivate rational clients to exchange keys.

Proofs of misbehavior (POMs) play an important role in BAR Gossip by ensuring that clients who send internally inconsistent messages risk eviction. Each message in our history and update exchanges includes a cryptographic hash of the previous message sent in that balanced exchange: if a client sends a briefcase whose contents differ from the agreed upon updates of the history exchange, then the history exchange messages plus the briefcase constitutes a POM.

We introduce a trusted agent of the broadcaster to audit possible POMs. In every round, this auditor polices the system by ordering a constant fraction of random clients to supply suspected POMs against other clients. Note that the auditor can decrypt the contents of well-formed briefcase messages because the broadcaster supplies the auditor with every client's private key. If a queried client does not have a POM against a peer, then the client must reply with a dummy message. We specify all audit responses to be of equal size, thereby removing a rational client's incentive to cover up POMs. The auditor treats clients that ignore audit requests as it would clients that have provably misbehaved. To reduce false positives because of transient network failures, the auditor allows sufficient time for a client to respond to audit requests. The auditor evicts a misbehaving client by sending a signed eviction notice to the broadcaster, who embeds all eviction notices in every update and stops sending updates to evicted clients. We discuss in Section 4.4 how to bound the overhead of eviction notices.

Our approach to making Balanced Exchange robust to rational deviations follows two principles: restricted choice and delayed gratification. Restricted choice provides Balanced Exchange's safety property: if a rational client decides to participate in an exchange, then that client sends only messages as prescribed by the protocol. Delayed gratification provides Balanced Exchange's liveness property: if links do not drop messages, then

two non-Byzantine clients participating in an exchange will complete that exchange. We delay gratification by postponing a rational client's receipt of useful updates until the last phase, key exchange. In the next section, we prove these properties hold under a reasonable set of assumptions.

4.2 Balanced Exchange Properties

We discuss Balanced Exchange's robustness against rational behavior within the framework of Nash equilibria [28]. In a Nash equilibrium, each client has a strategy and no client benefits from changing its strategy while the other clients keep their strategies unchanged. A limitation of Nash equilibria is that they do not consider rational clients that collude to increase collective utility. We empirically explore the impact of colluding rational clients in Section 5.4, but future work is needed to design gossip protocols with provable guarantees against such collusions.

In our analysis, we assume rational clients only consider strategies that maximize the utility of each exchange independent of concurrent or future exchanges. Our experiments demonstrate that this greedy strategy performs well in a streaming environment where there is a limited time to obtain useful updates. It is possible that more sophisticated strategies optimizing over multiple exchanges are superior to this myopic strategy; analyzing these strategies remains future work. The Balanced Exchange Protocol guarantees the following property:

Theorem 1. If two rational clients participating in a balanced exchange with each other seek to maximize the utility of that exchange independent of concurrent or future exchanges, then following the Balanced Exchange Protocol is a Nash equilibrium.

In the following sections, we show that each phase of a balanced exchange is a Nash equilibrium, implying that a balanced exchange is also a Nash equilibrium.

In each phase, we show that if a rational client assumes it peers are altruistic, then following Balanced Exchange is in that client's best interest. Note that the assumption that remaining clients are altruistic is an artifact of the Nash equilibria proof technique and is not a requirement of our protocol.

We simplify the presentation of the subsequent lemmas and proofs by treating any client that has issued a POM against itself as evicted. We use the following property regarding eviction in later proofs and include it here for reference.

Lemma 1. A rational client S i) never issues a POM against itself and ii) expects no benefit from communicating with evicted clients.

Proof Sketch: i) Because rational clients are cautious by nature, S does not issue a POM against itself for fear of being evicted.

ii) Suppose for contradiction that there exists an evicted client R for whom S expects positive benefit. Since S assumes all other clients are following the protocol correctly and a client can only be evicted by deviating from the protocol, R must be an altruistic client that deviates from the protocol, which is a contradiction.

4.2.1 Partner Selection

<u>Problem</u>: What if a rational client selects more partners per round than prescribed or biases its selections instead of choosing partners uniformly at random?

Partner selection highlights a fundamental difference between traditional gossip and gossip in the BAR setting. In a traditional gossip protocol, each client periodically selects a partner using a pseudo-random number generator (PRNG) and contacts that partner to request an exchange. Each client also accepts every request it receives. Random partner selection provides robustness against crashed clients, link failures, and targeted attacks. Yet, in a BAR model, the freedom to choose partners allows rational clients to select multiple partners not at random, thereby dissolving gossip's guarantees. In BAR Gossip, a client generates an unpredictable, deterministic seed for the PRNG to select a random partner, who can then verify and accept the selection using the same PRNG or reject it otherwise.

In BAR Gossip, a client S selects a partner for round rby seeding a PRNG with the signature $\langle r, BAL \rangle_S$. S then deterministically maps numbers generated by the PRNG to client ids until it finds the first partner R for which it does not have an eviction notice. This partner selection is deterministic, but unpredictable because no client other than S can generate S's signature for a seed value. As Figure 1 illustrates, to initiate a gossip request to R, S includes the seed and all eviction notices for clients that S could have selected before R. R then determines whether the seed is valid by verifying that i) the seed is a valid signature, ii) r is the current round, iii) all included eviction notices are valid, iv) the seeded PRNG generates R as the first non-evicted client, and v) this is the first time that S has presented this seed value to R. If all five tests pass, then R accepts the gossip request from S; otherwise, the seed is considered *invalid* and Raborts the exchange. This selection algorithm provides the following guarantee:

Lemma 2. Rational clients only send gossip requests to and accept gossip requests from clients as prescribed by valid seeds.

Proof Sketch: A rational client S may communicate with (a) the uniquely defined target sanctioned by the protocol, (b) an evicted client generated by the PRNG before any non-evicted client, or (c) some other client. By Lemma 1, (b) is not an option. Similarly, (c) is not an option either because S will not be wrongly contacted by an altruistic client nor would expect an altruistic client to engage in an exchange not sanctioned by the protocol. This leaves option (a) as the only feasible choice for a rational client.

We note that the argument against clients participating in unsanctioned exchanges is buttressed by the specific tangible concern that such exchanges would be done without the recourse of sending a POM to the auditor if either client in an exchange (quite rationally) were to cheat its partner by sending a different briefcase than the one agreed upon.

4.2.2 History Exchanges

<u>Problem</u>: What if a rational client lies about having (or not having) an update?

After a client S selects a partner R, they exchange histories—a history defines a set of update ids—using three messages. As Figure 1 illustrates, S provides in the first message a hash of its history H_S and the PRNG seed value (as discussed earlier) to R; the hash is a verifiable promise to send a given history. After verifying that S is entitled to communicate with R, R returns its current history H_R . In the final message, S divulges its actual history, H_S , to R who checks that the previously sent hash is consistent with the divulged history. Note that each client sends a history before learning its partner's history: S does so by sending a unique hash first and R by sending its actual history while possessing only an irreversible hash. This design promotes equal sharing, as neither client can tailor a history to its partner's. In particular, this design makes it difficult for a Byzantine client to maximize network traffic during the update exchange by sending a history that is the exact complement of its partner's.

Lemma 3. A rational client S does not divulge a history that does not match the original hash. Similarly, S terminates any exchange in which the divulged history does not match the original hash.

Proof Sketch: If S were to sign messages indicating different histories for the same exchange, then these messages constitute a POM and lead to eviction. By Lemma 1, S, being rational, would not do so.

Lemma 1 also ensures that S will not communicate with a client that divulges a history not matching the original hash because such actions would result in a POM.

Because update ids are public knowledge, a rational client S may still consider reporting a history $H_S' \neq H_S$ to increase the expected utility in an exchange. Applying the principle of *balanced cost* [1], we define all histories to be of fixed size, thereby removing any incentive S may have to save a few bytes by sending smaller histories. Therefore, the only way for S to obtain greater utility is by increasing the number of useful updates S receives in each exchange. We show that rational clients cannot increase expected utility by lying about histories.

Lemma 4. Consider a rational client S and an unexpired update $U \notin H_S$. If S participates in an exchange, then S reports a history H'_S such that $U \notin H'_S$.

Proof Sketch: Assume S deviates from the protocol by falsely claiming $U \in H_S'$. S does so if and only if it expects greater utility without risking eviction. In order to obtain this greater utility by falsely claiming $U \in H_S'$, S must send a briefcase message that claims to contain U. Since $U \notin H_S$, such a briefcase message is a POM. So according to Lemma 1, S does not send it.

Lemma 5. Consider a rational client S and an unexpired update $U \in H_S$. If S participates in an exchange, then S reports a history H_S' such that $U \in H_S'$.

Proof Sketch: Claiming $U \notin H'_S$ decreases the expected number of useful updates to be exchanged. Since a rational client deviates if and only if doing so increases expected utility, S would not claim $U \notin H'_S$.

4.2.3 Update Exchange

<u>Problem</u>: What if a rational client places fake or garbage data in briefcase messages?

After the history exchange commits S and R to sending the k most recent updates each possesses but the other lacks, S and R send the corresponding updates contained in signed briefcases. Each briefcase message contains i) the seed identifying this exchange, ii) a plaintext description of k update ids, and iii) the corresponding k updates encrypted with the hash of both the sender's private key and the exchange's seed value. The sender signs the briefcase, promising that the encrypted contents match the description. If either the received briefcase's seed value does not match the seed identifying this exchange or the briefcase's update list does not match the k expected updates, the receiver aborts the exchange without sending its decryption key.

Lemma 6. If a rational client S sends a briefcase message, then S includes the appropriate seed value and plaintext description for that exchange.

Proof Sketch: By Lemma 1, S will not include an inappropriate plaintext description, because the resulting briefcase message and history exchange messages constitute a POM. S will also include the appropriate seed value and signature because its partner R is unwilling to accept a briefcase message for which S does not fear being audited. \Box

S and R exchange decryption keys in the next phase. If a client receives a briefcase but not the corresponding decryption key, then the client includes the briefcase as a suspected POM for a future audit response.

Lemma 7. If a rational client S sends a briefcase message, then the encrypted contents correspond to the briefcase's plaintext description.

Proof Sketch: A briefcase whose contents differ from the plaintext description is a POM, which according to Lemma 1, S would never send.

4.2.4 Key Exchange

<u>Problem</u>: What if a rational node chooses not to send the key or sends an invalid key?

A client who is satisfied with its partner's briefcase enters the key exchange phase. In this phase, the client sends via UDP a key request containing the exchange seed and responds to key requests (also via UDP) with a signed response that contains *i*) the seed value and *ii*) the decryption key corresponding to the briefcase sent in the previous phase.

The deterministic fair exchange of decryption keys is impossible to solve without a trusted third party [32]. We show that in the setting of BAR Gossip, altruistic and rational clients can exchange keys *fairly enough* without a trusted third party. The linchpin in providing this is to use a *credible threat*. A client repeatedly sends key requests, up to some constant number of times, until it obtains a key response from its partner. Note that it is possible to tune the size of key requests to offset any asymmetry between download and upload capacity.

Lemma 8. If a rational client S responds to a key request, then S's response contains the appropriate symmetric key.

Proof Sketch: A key response whose contained key does not match the hash of the seed and sender's private key is a POM, which by Lemma 1, S does not send.

Lemma 9. If a rational client S is satisfied with a client R's briefcase, then S responds to R's key requests.

Proof Sketch: Assume S ignores R's key requests. S deviates because it expects greater utility from doing so. However, since R is following the protocol, then R quickly erodes S's increase in utility by making S receive multiple key requests.

Lemma 10. If a rational client S does not receive a key response from a client R, S will resend its key request.

Proof Sketch: If R is following the protocol, S reasons that the unreliability of UDP is responsible for the delay. S therefore resends the key request because deviating by keeping silent decreases S's expected utility.

4.2.5 Optimistic Push

The Optimistic Push Protocol provides a safety net for clients who have fallen behind by allowing clients to obtain missing updates without giving back a set of updates of equivalent value. Optimistic pushes follow the same structure as balanced exchanges. Partner selection is nearly identical. In round r, client S uses $\langle r, \mathsf{OPT} \rangle_S$ to seed the PRNG and ultimately selects a partner R in the same way as in the Balanced Exchange protocol.

The main difference between Balanced Exchange and Optimistic Push lies in what the parties disclose to each other during the history exchange and in how they determine the content of their respective briefcases during the update exchange. In particular, for the history exchange S forwards to R two lists: a young list, which contains the identifiers of some of the most recent updates S knows, and an old list, which contains the identifiers of updates that S is missing and that are about to expire. If R has nothing to offer from the old list, R terminates the exchange. Otherwise, R replies with a want list, which contains the identifiers of c updates from the young list that R is actually missing. S and R then exchange briefcases. S's briefcase contains the c updates from the want list with an appropriate plaintext description of the update ids. In contrast, R's briefcase contains the largest number, $b \le c$, of updates that R possesses from S's old list and c-b junk updates. Furthermore, the plaintext description on R's briefcase does not identify the exact contained updates, only that c items are inside, each of which can either be junk or from the old list. It is this flexibility to exchange deterministically generated junk data for good data that allows a receiver that has fallen behind to catch up.

We emphasize that junk updates are crafted to be larger than real updates. If junk updates were smaller, the Optimistic Push Protocol would encourage rational clients to deviate from the Balanced Exchange Protocol because updates might be had for cheaper in Optimistic Push. If junk updates were the same size as real updates, a rational client may prefer to send junk to maintain the scarcity of updates in that client's possession.

We regulate Optimistic Push with two parameters, push_age and push_size: the young list consists only of updates that have been broadcast within the last push_age rounds and push_size is an upper limit on the number of updates that the Receiver can place in its want list. Larger values of push_size help lagging clients catch up faster; however, they also increase the likelihood that such clients will waste bandwidth by sending junk.

The Optimistic Push Protocol follows nearly the same steps as the Balanced Exchange Protocol. As a result, the above lemmas (except Lemma 5) apply; a rational client may disingenuously claim to not have an update to reduce the expected number of received junk updates. Although restricted choice still limits the messages that a rational client will send in the Optimistic Push Protocol, the extra flexibility makes faithful participation less certain. For example, rational clients may choose to deviate from the Optimistic Push Protocol by simply not participating, never initiating pushes but responding to them, or sending junk updates in lieu of useful updates.

Although we cannot prove that a rational client would faithfully follow the Optimistic Push Protocol, our experimental evidence, in Section 5, suggests that a rational client obtains greater utility from following the protocol than from deviating.

4.3 Designing for Byzantine behavior

Byzantine behavior is a reality of distributed systems. While enticing rational clients to behave correctly in the presence of Byzantine behavior, we must also limit the negative impact of such behavior on good users of the system. In this paper we limit our attention to Byzantine nodes that exploit the messages and behaviors defined by our protocol.

In BAR Gossip, Byzantine nodes cannot subvert the system's safety properties. Because the broadcaster signs each update, a Byzantine node cannot tamper with the contents of any delivered update. With respect to the liveness property stated in Section 4.1, Byzantine nodes can impair progress by sending two types of messages: non-protocol messages and protocol messages. We regard generic DoS attacks based on non-protocol messages (e.g. bandwidth or connection flooding) as outside the scope of this protocol.

BAR Gossip is designed to be robust against protocolbased attacks on liveness even if initiated by a significant number of Byzantine clients. First, BAR Gossip's peer selection protocol limits the number of nodes that one can contact in a round—unlike traditional gossip, where a Byzantine node could potentially contact an unlimited number of nodes and involve them in useless exchanges. Second, Byzantine clients can inflict limited damage in the exchanges in which they participate. A Byzantine client can remain silent during an exchange to slow the spread of updates, but fortunately, gossip protocols are naturally resilient to crash failures. One remaining concern is that a Byzantine client could impact liveness by luring its partners into expensive message exchanges that do not eventually result in the dissemination of useful updates. We explore this kind of attack in Section 5.5, where we show that altruistic clients still deliver over 93% of updates in a timely manner even when 20% of the clients are Byzantine.

4.4 Optimizations

To increase the practicality of BAR Gossip, we incorporate four optimizations. First, to prevent a client from being overwhelmed by valid gossip requests in a round, we use the standard heuristic that each client accepts requests up to some per round maximum and ignores further requests that round [4,8]. Second, to prevent spikes in used bandwidth, each client in a balanced exchange limits the number of updates that are actually swapped, similar to the Round Retransmission Limit optimization of [4], by including this limit in history exchanges. Third, the broadcaster embeds each eviction notice into a constant number of updates, thereby bounding the overhead of each eviction. With high probability, every client learns of an eviction within deadline rounds. Fourth, clients elide eviction notices that are older than deadline rounds from gossip requests.

5 Evaluation

In this section, we show that BAR Gossip is a robust p2p streaming protocol capable of providing stable and reliable throughput. We evaluate BAR Gossip through experiments and simulations—we denote figures derived from simulation data with "[sim]." Our evaluation demonstrates that BAR Gossip:

- Outperforms traditional gossip in the presence of rational clients
- 2. Prevents unilateral rational deviation
- 3. Is stable in the presence of significant collusion
- 4. Tolerates up to 20% of the clients being Byzantine

5.1 Methodology

Several parameters regulate the Balanced Exchange and Optimistic Push Protocols. The broadcaster multicasts ups_per_round updates per round and sends each update to nSeeds random clients. Each update expires deadline rounds after it was multicast. In optimistic pushes, push_age denotes the maximum age of updates sent in the young list, while push_size is the maximum length of the want list. The ratio of junk update

Protocol Parameter	Simulation	Prototype
ups_per_round(updates)	10	98-101
nSeeds(clients)	12	3
deadline(rounds)	10	10
push_size(updates)	2	20
push_age(updates)	3	3
junk_cost	2	1.39
# clients	250	45

Table 1: Parameter settings used in simulations and prototype experiments.

size to real update size is junk_cost> 1. Table 1 provides the values for these parameters for our simulation and prototype experiments. We use lower parameter settings for the simulation, so that our simulator would terminate in a reasonable amount of time. Note that we maintain approximately the same ratio of push_size to ups_per_round in both settings.

For our prototype evaluations, we implement BAR Gossip in Python to stream an MPEG-4 video [34]. We recorded a 200 Kbps UDP video stream at 30 frames per second using Quicktime Broadcaster with one key frame every 60 frames. Quicktime Broadcaster generates UDP datagrams for the broadcast with an average size of 179 bytes ($\sigma=62$), resulting in 116–131 datagrams per second.

Our broadcaster, auditor, and clients are a mix of 45 600 MHz and 850 MHz Emulab machines sharing a 100 Mbps Ethernet subnet, configured with a 100ms endto-end latency and 1% probability of any packet being dropped. The broadcaster reads the recorded video from disk, encapsulates on average three UDP datagrams into an update, pads every update to the same size (640 bytes), and unicasts each update using UDP to a random five clients. Clients then exchange updates as in Figure 1. A client delivers an update by extracting the contained datagrams and sending them to the local Quicktime client that displays the video content. We use MD5 to compute cryptographic hashes, RSA with full domain hashing [3] to create unique signatures and the Mersenne Twister algorithm [24] to generate pseudo-random numbers. Each client used on average 299 Kbps of upload bandwidth.

In the following sections we measure the reliability (expressed as the percentage of updates received by the deadline), jitter (measured as the percentage of rounds in which any update missed its deadline), and bandwidth characteristics of BAR Gossip. Unless otherwise noted, measurements in simulations are averaged over 1000 rounds and using the prototype are averaged over 180 rounds across 15 trials. Error bars are small in our data and elided from graphs for clarity.

5.2 Traditional Gossip

We now compare BAR Gossip against a traditional pushpull gossip protocol [8], where each client following the

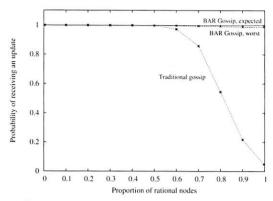


Figure 2: [sim] Reliability experienced by an altruistic client using traditional gossip versus BAR Gossip.

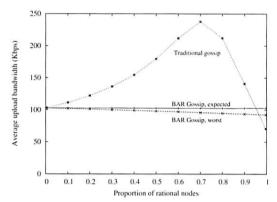


Figure 3: [sim] Send bandwidth used by an altruistic client using traditional gossip versus BAR Gossip.

protocol randomly selects one partner per round, exchanges histories, and then exchanges missing updates. We demonstrate that this traditional protocol is ill-suited for a BAR environment even when no client is Byzantine and every rational client initiates only one exchange per round. The intuition here is that a rational client maximizes its utility by not sending updates. Altruistic clients, therefore, do all the work of disseminating information in the system.

Figures 2 and 3 plot the reliability seen and bandwidth used, respectively, by an altruistic client as the proportion of rational clients in the system increases. While BAR Gossip's lines remain relatively constant in both graphs, traditional gossip's degrades noticeably.

We include two lines for BAR Gossip because although we can prove a rational client will faithfully participate in balanced exchanges, we do not have a similar proof for optimistic pushes. The "BAR Gossip, expected" line plots the best reliability an altruistic client can see using BAR Gossip, corresponding to the case where rational clients faithfully participate in optimistic pushes. We label the line "expected" because our empirical data suggests rational clients obtain greatest utility by actually following the Optimistic Push Protocol. The

Strategy	Accepts OP	Initiates OP	Returns
Proactive/Data	Yes	Yes	Data
Proactive/Junk	Yes	Yes	Junk
Proactive/Decline	No	Yes	None
Passive/Data	Yes	No	Data
Passive/Junk	Yes	No	Junk
Passive/Decline	No	No	None

Table 2: Six strategies a rational client may follow with regards to the Optimistic Push Protocol.

"BAR Gossip, worst" line represents the case where rational clients never initiate optimistic pushes and send as much junk in briefcases as the protocol allows. Among the strategies we consider (see Section 5.3), this second strategy yields the worst reliability for an altruistic client.

Figure 2 shows that a client following BAR Gossip receives almost all updates even when all other clients are rational. In contrast, altruistic clients in traditional gossip experience significantly lower reliability once the number of rational clients exceeds 50%. When all clients but one are rational, traditional gossip provides only the reliability that the broadcaster can guarantee alone.

Figure 3 illustrates that in BAR Gossip an altruistic client's consumed bandwidth is nearly independent of the proportion of rational clients. Traditional gossip, on the other hand, requires altruistic clients to shoulder the entire burden of spreading updates, with bandwidth spiking sharply when rational clients account for 70% of the system, before tumbling down to almost nothing. This dramatic fall coincides with the sharp decline in reliability in Figure 2. In these areas of the graphs, rational clients receive the majority of multicast updates and decline to spread them, reducing both reliability and bandwidth devoted to gossiped data.

5.3 Unilateral Rational Deviation

We now examine deviant strategies that a rational client might pursue. In our experiments, a rational client pursues these strategies while the remaining clients continue to follow the protocol as specified. Note that this is the experimental analog to the standard Nash equilibrium proof technique.

In this analysis, we make two simplifying assumptions. First, a rational client's primary concern is to improve the delivered stream's quality by maximizing reliability and minimizing jitter; minimizing consumed bandwidth is a subordinate goal. Second, a rational client missing one or more updates always expects positive utility from participating in a balanced exchange. Under this second assumption, rational clients faithfully execute the Balanced Exchange Protocol. We now consider the choices available to a rational client with respect to Optimistic Push.

Table 2 lists the five strategies we consider that a ra-

Strategy	Avg. Jitter	Std. Deviation
Proactive/Data	0.48%	1.16%
Proactive/Junk	0.32%	0.78%
Proactive/Decline	11.59%	6.22%
Passive/Data	18.10%	6.08%
Passive/Junk	14.76%	9.44%
Passive/Decline	47.94%	7.52%

Table 3: Rational client's experienced jitter pursuing different strategies.

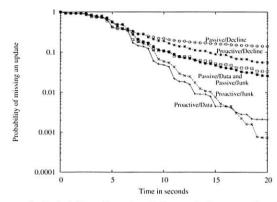


Figure 4: Probability of a rational client missing an update for different strategies.

tional client may pursue to deviate from the Optimistic Push Protocol. *Proactive* strategies dictate that a rational client initiates optimistic pushes as specified by the Optimistic Push Protocol. In contrast, *passive* strategies specify to never initiate optimistic pushes. *Data, junk*, and *decline* strategies prescribe that rational clients responding to an optimistic push send useful updates (when possible), send as much junk as allowed, or decline the exchange, respectively. Note that following the Optimistic Push Protocol corresponds to the the Proactive/Data strategy.

Figure 4 shows for each of the six strategies the probability that the rational client will miss an update, where lower lines correspond to better reliability. Table 3 provides the corresponding jitter for each strategy. When taken together, Figure 4 and Table 3 imply that rational clients will follow either proactive/data or proactive/junk strategies. This is perhaps not surprising, given that proactive strategies perform additional exchanges that are likely to result in more deliverable updates than passive strategies.

The tie breaker between the top two strategies comes from Figure 5, in which proactive/data uses an average of 300 Kbps of upload bandwidth compared against proactive/junk's 317 Kbps. This is not an accident: we have *designed* BAR Gossip with junk_cost> 1 so that rational clients prefer filling their briefcases with valuable updates, rather than junk, whenever possible.

The conclusion we draw from this set of experiments

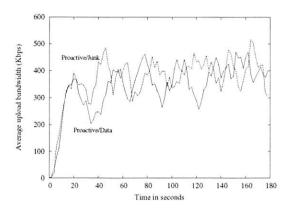


Figure 5: Rational client's consumed bandwidth for different strategies.

is that a rational client, when surrounded by other clients that follow BAR Gossip, has no obvious incentive for deviation-in fact, quite the contrary. While our experiments clearly fall short of proving that BAR Gossip as a whole (Balanced Exchange plus Optimistic Push) constitutes a Nash equilibrium, it does suggest that a Nash equilibrium is likely to be found at or near the strategy that corresponds to BAR Gossip. For instance, while we are unable to prove that there are no beneficial hybrid strategies that, depending on the environment, switch between two or more of the the six strategies we have considered, it appears that the benefit of a proactive strategy derives from consistently participating in more exchanges, making it unlikely that switching occasionally to a passive strategy would provide a net gain. As for switching among proactive strategies, it yields no change in benefit while changing bandwidth costs, also providing little room for improvement.

Overall, we believe that the expected and worst case lines in Figures 2 and 3 provide a reasonable bound on the actual behavior of rational clients in BAR Gossip, and that the likely behavior is near the expected line.

5.4 Rational Collusion

We now explore the effect of multiple rational clients coordinating their actions to maximize their collective utility. We perform a series of simulations to assess the impact such a group may have on clients following the protocol. A complete analysis that defines optimal collusion strategies is future work.

We assume that colluding and non-colluding rational clients share a utility function. We also assume that colluding clients run a private protocol to disseminate updates among themselves. This protocol may be an alternative BAR protocol or it may be a non-BAR protocol bolstered by a high level of trust among colluding clients. We simulate a *perfect collusion* scenario in which every colluding client immediately broadcasts new updates

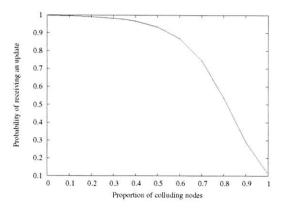


Figure 6: [sim] Effect of collusion on an altruistic client's reliability when colluding clients are following the passive/decline strategy.

within the group at no cost. This source of updates reduces the incentive to fully participate in the BAR Gossip protocol. In particular, colluding clients only run the Balanced Exchange protocol.

Figure 6 shows how the size of a perfect collusion group affects the quality of the stream seen by a client following BAR Gossip. The intuition for the degraded performance is *i*) a non-colluding client trades little when participating in a balanced exchange with a colluding client and *ii*) colluding clients do not participate in optimistic pushes. In perfect collusion groups, colluding clients get most of their updates for free from other colluding clients, reducing their contributions to the rest of the system.

We find that when the collusion group size reaches 50% of the participants, altruistic clients see an average convergence of 93% for an update, resulting in an unusable stream. Although near-perfect collusion among small groups seems plausible, it is unclear that collusion on a large scale is a significant threat. As the colluding group grows, so do the challenges of coordinating and trusting clients. Ironically, as a colluding group grows, it might require BAR Gossip to distribute updates internally as trust begins to break down among members.

5.5 Byzantine Deviation

Rational players behave according to a well-known utility function and represent most of the clients in a p2p system across multiple administrative domains. A few clients, however, may possess unknown utility functions or behave arbitrarily due to ill-will or malfunction, possibly affecting rational behavior [1]. Note that these Byzantine clients may be disinterested in stream packets and may prefer maximizing damage irrespective of consumed bandwidth, allowing strategies like DoS attacks.

To assess BAR Gossip's robustness to Byzantine participation, we explore one malicious goal that a Byzan-

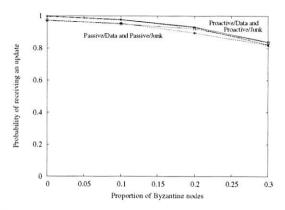


Figure 7: Rational client's reliability for different strategies.

tine client could possess. We assume that this goal is the inverse of any rational player's: to increase the cost and decrease the benefit of all rational clients. We consider Byzantine behavior within the limits of the BAR Gossip protocol, which, as discussed in Section 4.3, limits Byzantine clients to degrading performance only through their own exchanges.

In the following experiments, a Byzantine client provides a history during a balanced exchange that is the complement of its partner's to induce the other client to exchange the maximum number of updates. During an optimistic exchange, a Byzantine client always announces a complete young list and empty old list if initiating, and requests the entire young list if receiving. A Byzantine client never enters the update or key exchange phases, so as not to generate a POM and risk eviction, but still inducing its partner to devote significant bandwidth to the exchange without receiving any benefit. The presence of Byzantine clients can be viewed as an increase in the overhead associated with the environment as the costs associated with Byzantine clients depends upon the probability of entering an exchange with a Byzantine client. To show worst case behavior under this attack, non-Byzantine clients in our experiments ignore previous unproductive exchanges. We elide proactive/decline and passive/decline strategies in which rational clients decline to participate in optimistic pushes.

Figures 7 and 8 show the reliability seen and bandwidth used, respectively, by a rational client pursuing each strategy in the presence of different proportions of Byzantine clients. The remaining non-Byzantine clients are altruistic. The choice of strategies is similar to Section 5.3 where we considered unilateral deviation with no Byzantine clients. Passive and proactive strategies deliver unwatchable video streams when the proportion of Byzantine clients reaches 10% and 30%, respectively.

We conclude that among the strategies available, a rational client should follow the protocol (proactive/data) regardless of the presence of Byzantine clients. If all

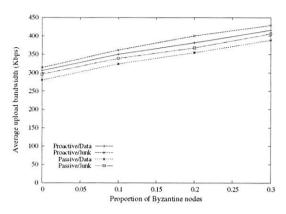


Figure 8: Rational client's consumed bandwidth for different strategies.

non-Byzantine clients are following the protocol, with a system comprised of 20% Byzantine clients, the bandwidth costs remain relatively constant while the convergence suffers by less than 7%.

6 Conclusion

We present the first peer-to-peer data streaming application with predictable throughput and low latency in the presence of correct, selfish, and malicious nodes. At the core of our application is BAR Gossip, the first gossip protocol defined under the BAR model. We leverage a unique signature scheme to generate verifiable pseudorandom numbers, allowing us to eliminate the opportunity for rational nodes to hide behind nondeterminism without sacrificing the benefits of the random communication pattern of gossip. Our experiments and simulations show that our protocol provides good convergence properties as long as no more than 20% of the nodes are Byzantine or no more than 40% of the nodes collude. In both cases, nodes following the protocol receive more than 93% of updates in a timely manner.

7 Acknowledgements

The authors would like to thank the anonymous reviewers and Brad Chen for shepherding our paper. We would also like to thank Jean-Phillipe Martin, Vitaly Shmatikov and Peter Stone for helpful discussions about game theory, signature schemes, and fair exchange.

This work was supported in part by NSF award CNS 0509338 and NSF CyberTrust award 0430510.

References

- A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR Fault Tolerance for Cooperative Services. In *Proc.* 20th SOSP, Brighton, UK, Oct. 2005. ACM Press.
- [2] G. Badishi, I. Keidar, and A. Sasson. Exposing and eliminating vulnerabilities to denial of service attacks in secure gossip-based multicast. In *Proc. DSN-2004*, page 223, Washington, DC, USA, 2004. IEEE Computer Society.
- [3] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In CCS '93: Proceedings of the 1st ACM conference on Computer and communications security, pages 62–73, New York, NY, USA, 1993. ACM Press.
- [4] K. P. Birman, M. Hayden, O. Oskasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. ACM Trans. Comput. Syst., 17(2):41–88, May 1999.
- [5] K. P. Birman, R. van Renesse, and W. Vogels. Spinglass: Secure and scalable communications tools for mission-critical computing. In *DARPA DISCEX-2001*, 2001.
- [6] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *Proc. 19th SOSP*, pages 298–313. ACM Press, 2003.
- [7] B. Cohen. Incentives build robustness in BitTorrent. In Proc. 2nd IPTPS, 2003.
- [8] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. 11th SOSP*, Aug. 1987.
- [9] A. K. Dixit and S. Skeath. *Games of Strategy*. W. W. Norton & Company, 1999.
- [10] J. R. Douceur. The Sybil attack. In *Proc. 1st IPTPS*, pages 251–260. Springer-Verlag, 2002.
- [11] P. Eugster, S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proc. DSN*-2001, pages 443–452, July 2001.
- [12] M. Feldman, C. Papadimitriou, J. Chuang, and I. Stoica. Freeriding and whitewashing in peer-to-peer systems. In *Proc. PINS*, pages 228–236. ACM Press, 2004.
- [13] A. J. Ganesh, A.-M. Kermarrec, and L. Massouli. Peer-topeer membership management for gossip-based protocols. *IEEE Trans. Comput.*, 52(2):139–149, 2003.
- [14] I. Gupta, K. Birman, and R. van Renesse. Fighting fire with fire: using randomized gossip to combat stochastic scalability limits. *Journal of Quality and Reliability Engineering Interna*tional, 18(3):165–184, 2002.
- [15] A. Habib and J. Chuang. Incentive mechanism for peer-to-peer media streaming. In 12th IEEE International Workshop on Quality of Service., 2004.
- [16] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The Eigentrust algorithm for reputation management in P2P networks. In WWW03, pages 640–651, New York, NY, USA, 2003. ACM Press.
- [17] S. Kremer, O. Markowitch, and J. Zhou. An intensive survey of non-repudiation protocols. *Computer Communications*, 25(17):1606–1621, Nov. 2002.
- [18] D. Levin, R. Sherwood, and B. Bhattacharjee. Fair file swarming with FOX. In *Proc. 5th IPTPS*, Feb 2006.
- [19] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. In *European Dependable Computing Conference*, pages 364–379, 1999.
- [20] D. Malkhi, Y. Mansour, and M. K. Reiter. Diffusion without false rumors: on propagating updates in a byzantine environment. *Theor. Comput. Sci.*, 299(1-3):289–306, 2003.
- [21] D. Malkhi, E. Pavlov, and Y. Sella. Optimal unconditional information diffusion. In *Proc. 15th DISC*, pages 63–77, London, UK, 2001. Springer-Verlag.
- [22] D. Malkhi, M. Reiter, O. Rodeh, and Y. Sella. Efficient update

- diffusion in Byzantine environments. In Proc. 20th SRDS, 2001.
- [23] S. Marti and H. Garcia-Molina. Identity crisis: Anonymity vs. reputation in p2p systems. page 134. IEEE Computer Society, 2003.
- [24] M. Matsumoto and T. Nishimura. Mersenne twister: a 623dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul., 8(1):3–30, 1998.
- [25] R. Melamed and I. Keidar. Araneola: A scalable reliable multicast system for dynamic environments. In *Proc. 3rd NCA*, pages 5–14, Washington, DC, USA, 2004. IEEE Computer Society.
- [26] Y. Minsky and F. Schneider. Private communication.
- [27] A. Nandi, T.-W. J. Ngan, A. Singh, P. Druschel, and D. S. Wallach. Scrivener: Providing incentives in cooperative content distribution systems. In *Proc. 6th Middleware*, Grenoble, France, Nov. 2005.
- [28] J. Nash. Non-cooperative games. The Annals of Mathematics, 54:286–295, Sept 1951.
- [29] National Institute of Standards and Technology. FIPS PUB 186-2: Digital Signature Standard (DSS). Jan. 2000.
- [30] T.-W. Ngan, D. S. Wallach, and P. Druschel. Incentivescompatible peer-to-peer multicast. In 2nd Workshop on Economics of Peer-to-Peer Systems, 2004.
- [31] D. C. Oppen and Y. K. Dalal. The clearinghouse: a decentralized agent for locating named objects in a distributed environment. ACM Trans. Inf. Syst., 1(3):230–253, 1983.
- [32] H. Pagnia and F. C. Gärtner. On the impossibility of fair exchange without a trusted third party. Technical Report TUD-BS-1999-02, Darmstadt University of Technology, Department of Computer Science, Darmstadt, Germany, Mar. 1999.
- [33] D. C. Parkes. Iterative Combinatorial Auctions: Achieving Economic and Computational Efficiency. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, May 2001.
- [34] A. Puri and A. Eleftheriadis. MPEG-4: an object-based multimedia coding standard supporting mobile applications. *Mob. Netw. Appl.*, 3(1):5–32, 1998.
- [35] R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical report, Ithaca, NY, USA, 1998.
- [36] P. Resnick, K. Kuwabara, R. Zeckhauser, and E. Friedman. Reputation systems. Commun. ACM, 43(12):45–48, 2000.
- [37] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th SOSP*, pages 188–201. ACM Press, 2001.
- [38] R. van Renesse, K. P. Birman, D. Dumitriu, and W. Vogels. Scalable management and data mining using Astrolabe. In *Proc. 1st IPTPS*, pages 280–294, London, UK, 2002. Springer-Verlag.
- [39] R. van Renesse, H. Johansen, and A. Allavena. Fireflies: Scalable support for intrusion-tolerant overlay networks. In *EuroSys* '06, 2006.
- [40] W. Vogels, R. van Renesse, and K. Birman. The power of epidemics: robust communication for large-scale distributed systems. SIGCOMM Comput. Commun. Rev., 33(1):131–135, 2003.
- [41] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. Sybil-guard: Defending against sybil attacks via social networks. In ACM SIGCOMM '06, Sept.
- [42] X. Zhang, J. Liu, B. Li, and T. P. Yum. DONet/CoolStreaming: A data-driven overlay network for live media streaming. In *IEEE INFOCOM*, Mar. 2005.

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

2 Data Model

A Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

(row:string, column:string, time:int64) → string

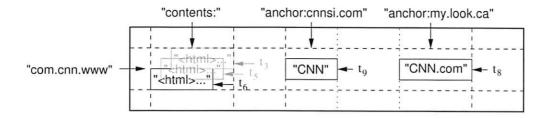


Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps t_3 , t_5 , and t_6 .

We settled on this data model after examining a variety of potential uses of a Bigtable-like system. As one concrete example that drove some of our design decisions, suppose we want to keep a copy of a large collection of web pages and related information that could be used by many different projects; let us call this particular table the *Webtable*. In Webtable, we would use URLs as row keys, various aspects of web pages as column names, and store the contents of the web pages in the contents: column under the timestamps when they were fetched, as illustrated in Figure 1.

Rows

The row keys in a table are arbitrary strings (currently up to 64KB in size, although 10-100 bytes is a typical size for most of our users). Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row), a design decision that makes it easier for clients to reason about the system's behavior in the presence of concurrent updates to the same row.

Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned. Each row range is called a *tablet*, which is the unit of distribution and load balancing. As a result, reads of short row ranges are efficient and typically require communication with only a small number of machines. Clients can exploit this property by selecting their row keys so that they get good locality for their data accesses. For example, in Webtable, pages in the same domain are grouped together into contiguous rows by reversing the hostname components of the URLs. For example, we store data for maps.google.com/index.html under the key com.google.maps/index.html. Storing pages from the same domain near each other makes some host and domain analyses more efficient.

Column Families

Column keys are grouped into sets called *column families*, which form the basic unit of access control. All data stored in a column family is usually of the same type (we compress data in the same column family together). A column family must be created before data can be stored under any column key in that family; after a family has been created, any column key within the family can be used. It is our intent that the number of distinct column families in a table be small (in the hundreds at most), and that families rarely change during operation. In contrast, a table may have an unbounded number of columns.

A column key is named using the following syntax: family:qualifier. Column family names must be printable, but qualifiers may be arbitrary strings. An example column family for the Webtable is language, which stores the language in which a web page was written. We use only one column key in the language family, and it stores each web page's language ID. Another useful column family for this table is anchor; each column key in this family represents a single anchor, as shown in Figure 1. The qualifier is the name of the referring site; the cell contents is the link text.

Access control and both disk and memory accounting are performed at the column-family level. In our Webtable example, these controls allow us to manage several different types of applications: some that add new base data, some that read the base data and create derived column families, and some that are only allowed to view existing data (and possibly not even to view all of the existing families for privacy reasons).

Timestamps

Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp. Bigtable timestamps are 64-bit integers. They can be assigned by Bigtable, in which case they represent "real time" in microseconds, or be explicitly assigned by client

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation rl(T, "com.cnn.www");
rl.Set("anchor:www.c-span.org", "CNN");
rl.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &rl);
```

Figure 2: Writing to Bigtable.

applications. Applications that need to avoid collisions must generate unique timestamps themselves. Different versions of a cell are stored in decreasing timestamp order, so that the most recent versions can be read first.

To make the management of versioned data less onerous, we support two per-column-family settings that tell Bigtable to garbage-collect cell versions automatically. The client can specify either that only the last *n* versions of a cell be kept, or that only new-enough versions be kept (e.g., only keep values that were written in the last seven days).

In our Webtable example, we set the timestamps of the crawled pages stored in the contents: column to the times at which these page versions were actually crawled. The garbage-collection mechanism described above lets us keep only the most recent three versions of every page.

3 API

The Bigtable API provides functions for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as access control rights.

Client applications can write or delete values in Bigtable, look up values from individual rows, or iterate over a subset of the data in a table. Figure 2 shows C++ code that uses a RowMutation abstraction to perform a series of updates. (Irrelevant details were elided to keep the example short.) The call to Apply performs an atomic mutation to the Webtable: it adds one anchor to www.cnn.com and deletes a different anchor.

Figure 3 shows C++ code that uses a Scanner abstraction to iterate over all anchors in a particular row. Clients can iterate over multiple column families, and there are several mechanisms for limiting the rows, columns, and timestamps produced by a scan. For example, we could restrict the scan above to only produce anchors whose columns match the regular expression anchor:*.cnn.com, or to only produce anchors whose timestamps fall within ten days of the current time.

Figure 3: Reading from Bigtable.

Bigtable supports several other features that allow the user to manipulate data in more complex ways. First, Bigtable supports single-row transactions, which can be used to perform atomic read-modify-write sequences on data stored under a single row key. Bigtable does not currently support general transactions across row keys, although it provides an interface for batching writes across row keys at the clients. Second, Bigtable allows cells to be used as integer counters. Finally, Bigtable supports the execution of client-supplied scripts in the address spaces of the servers. The scripts are written in a language developed at Google for processing data called Sawzall [28]. At the moment, our Sawzall-based API does not allow client scripts to write back into Bigtable, but it does allow various forms of data transformation, filtering based on arbitrary expressions, and summarization via a variety of operators.

Bigtable can be used with MapReduce [12], a framework for running large-scale parallel computations developed at Google. We have written a set of wrappers that allow a Bigtable to be used both as an input source and as an output target for MapReduce jobs.

4 Building Blocks

Bigtable is built on several other pieces of Google infrastructure. Bigtable uses the distributed Google File System (GFS) [17] to store log and data files. A Bigtable cluster typically operates in a shared pool of machines that run a wide variety of other distributed applications, and Bigtable processes often share the same machines with processes from other applications. Bigtable depends on a cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures, and monitoring machine status.

The Google *SSTable* file format is used internally to store Bigtable data. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Operations are provided to look up the value associated with a specified

key, and to iterate over all key/value pairs in a specified key range. Internally, each SSTable contains a sequence of blocks (typically each block is 64KB in size, but this is configurable). A block index (stored at the end of the SSTable) is used to locate blocks; the index is loaded into memory when the SSTable is opened. A lookup can be performed with a single disk seek: we first find the appropriate block by performing a binary search in the in-memory index, and then reading the appropriate block from disk. Optionally, an SSTable can be completely mapped into memory, which allows us to perform lookups and scans without touching disk.

Bigtable relies on a highly-available and persistent distributed lock service called Chubby [8]. A Chubby service consists of five active replicas, one of which is elected to be the master and actively serve requests. The service is live when a majority of the replicas are running and can communicate with each other. Chubby uses the Paxos algorithm [9, 23] to keep its replicas consistent in the face of failure. Chubby provides a namespace that consists of directories and small files. Each directory or file can be used as a lock, and reads and writes to a file are atomic. The Chubby client library provides consistent caching of Chubby files. Each Chubby client maintains a session with a Chubby service. A client's session expires if it is unable to renew its session lease within the lease expiration time. When a client's session expires, it loses any locks and open handles. Chubby clients can also register callbacks on Chubby files and directories for notification of changes or session expiration.

Bigtable uses Chubby for a variety of tasks: to ensure that there is at most one active master at any time; to store the bootstrap location of Bigtable data (see Section 5.1); to discover tablet servers and finalize tablet server deaths (see Section 5.2); to store Bigtable schema information (the column family information for each table); and to store access control lists. If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable. We recently measured this effect in 14 Bigtable clusters spanning 11 Chubby instances. The average percentage of Bigtable server hours during which some data stored in Bigtable was not available due to Chubby unavailability (caused by either Chubby outages or network issues) was 0.0047%. The percentage for the single cluster that was most affected by Chubby unavailability was 0.0326%.

5 Implementation

The Bigtable implementation has three major components: a library that is linked into every client, one master server, and many tablet servers. Tablet servers can be

dynamically added (or removed) from a cluster to accomodate changes in workloads.

The master is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing tablet-server load, and garbage collection of files in GFS. In addition, it handles schema changes such as table and column family creations.

Each tablet server manages a set of tablets (typically we have somewhere between ten to a thousand tablets per tablet server). The tablet server handles read and write requests to the tablets that it has loaded, and also splits tablets that have grown too large.

As with many single-master distributed storage systems [17, 21], client data does not move through the master: clients communicate directly with tablet servers for reads and writes. Because Bigtable clients do not rely on the master for tablet location information, most clients never communicate with the master. As a result, the master is lightly loaded in practice.

A Bigtable cluster stores a number of tables. Each table consists of a set of tablets, and each tablet contains all data associated with a row range. Initially, each table consists of just one tablet. As a table grows, it is automatically split into multiple tablets, each approximately 100-200 MB in size by default.

5.1 Tablet Location

We use a three-level hierarchy analogous to that of a B⁺-tree [10] to store tablet location information (Figure 4).

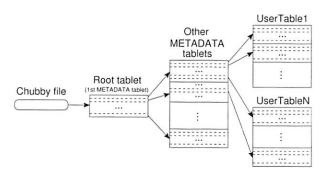


Figure 4: Tablet location hierarchy.

The first level is a file stored in Chubby that contains the location of the *root tablet*. The *root tablet* contains the location of all tablets in a special METADATA table. Each METADATA tablet contains the location of a set of user tablets. The *root tablet* is just the first tablet in the METADATA table, but is treated specially—it is never split—to ensure that the tablet location hierarchy has no more than three levels.

The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet's table

identifier and its end row. Each METADATA row stores approximately 1KB of data in memory. With a modest limit of 128 MB METADATA tablets, our three-level location scheme is sufficient to address 2^{34} tablets (or 2^{61} bytes in 128 MB tablets).

The client library caches tablet locations. If the client does not know the location of a tablet, or if it discovers that cached location information is incorrect, then it recursively moves up the tablet location hierarchy. If the client's cache is empty, the location algorithm requires three network round-trips, including one read from Chubby. If the client's cache is stale, the location algorithm could take up to six round-trips, because stale cache entries are only discovered upon misses (assuming that METADATA tablets do not move very frequently). Although tablet locations are stored in memory, so no GFS accesses are required, we further reduce this cost in the common case by having the client library prefetch tablet locations: it reads the metadata for more than one tablet whenever it reads the METADATA table.

We also store secondary information in the METADATA table, including a log of all events pertaining to each tablet (such as when a server begins serving it). This information is helpful for debugging and performance analysis.

5.2 Tablet Assignment

Each tablet is assigned to one tablet server at a time. The master keeps track of the set of live tablet servers, and the current assignment of tablets to tablet servers, including which tablets are unassigned. When a tablet is unassigned, and a tablet server with sufficient room for the tablet is available, the master assigns the tablet by sending a tablet load request to the tablet server.

Bigtable uses Chubby to keep track of tablet servers. When a tablet server starts, it creates, and acquires an exclusive lock on, a uniquely-named file in a specific Chubby directory. The master monitors this directory (the servers directory) to discover tablet servers. A tablet server stops serving its tablets if it loses its exclusive lock: e.g., due to a network partition that caused the server to lose its Chubby session. (Chubby provides an efficient mechanism that allows a tablet server to check whether it still holds its lock without incurring network traffic.) A tablet server will attempt to reacquire an exclusive lock on its file as long as the file still exists. If the file no longer exists, then the tablet server will never be able to serve again, so it kills itself. Whenever a tablet server terminates (e.g., because the cluster management system is removing the tablet server's machine from the cluster), it attempts to release its lock so that the master will reassign its tablets more quickly.

The master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassigning those tablets as soon as possible. To detect when a tablet server is no longer serving its tablets, the master periodically asks each tablet server for the status of its lock. If a tablet server reports that it has lost its lock, or if the master was unable to reach a server during its last several attempts, the master attempts to acquire an exclusive lock on the server's file. If the master is able to acquire the lock, then Chubby is live and the tablet server is either dead or having trouble reaching Chubby, so the master ensures that the tablet server can never serve again by deleting its server file. Once a server's file has been deleted, the master can move all the tablets that were previously assigned to that server into the set of unassigned tablets. To ensure that a Bigtable cluster is not vulnerable to networking issues between the master and Chubby, the master kills itself if its Chubby session expires. However, as described above, master failures do not change the assignment of tablets to tablet servers.

When a master is started by the cluster management system, it needs to discover the current tablet assignments before it can change them. The master executes the following steps at startup. (1) The master grabs a unique *master* lock in Chubby, which prevents concurrent master instantiations. (2) The master scans the servers directory in Chubby to find the live servers. (3) The master communicates with every live tablet server to discover what tablets are already assigned to each server. (4) The master scans the METADATA table to learn the set of tablets. Whenever this scan encounters a tablet that is not already assigned, the master adds the tablet to the set of unassigned tablets, which makes the tablet eligible for tablet assignment.

One complication is that the scan of the METADATA table cannot happen until the METADATA tablets have been assigned. Therefore, before starting this scan (step 4), the master adds the root tablet to the set of unassigned tablets if an assignment for the root tablet was not discovered during step 3. This addition ensures that the root tablet will be assigned. Because the root tablet contains the names of all METADATA tablets, the master knows about all of them after it has scanned the root tablet.

The set of existing tablets only changes when a table is created or deleted, two existing tablets are merged to form one larger tablet, or an existing tablet is split into two smaller tablets. The master is able to keep track of these changes because it initiates all but the last. Tablet splits are treated specially since they are initiated by a tablet server. The tablet server commits the split by recording information for the new tablet in the METADATA table. When the split has committed, it notifies the master. In case the split notification is lost (either

because the tablet server or the master died), the master detects the new tablet when it asks a tablet server to load the tablet that has now split. The tablet server will notify the master of the split, because the tablet entry it finds in the METADATA table will specify only a portion of the tablet that the master asked it to load.

5.3 Tablet Serving

The persistent state of a tablet is stored in GFS, as illustrated in Figure 5. Updates are committed to a commit log that stores redo records. Of these updates, the recently committed ones are stored in memory in a sorted buffer called a *memtable*; the older updates are stored in a sequence of SSTables. To recover a tablet, a tablet server

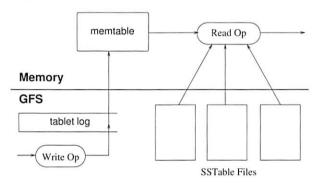


Figure 5: Tablet Representation

reads its metadata from the METADATA table. This metadata contains the list of SSTables that comprise a tablet and a set of a redo points, which are pointers into any commit logs that may contain data for the tablet. The server reads the indices of the SSTables into memory and reconstructs the memtable by applying all of the updates that have committed since the redo points.

When a write operation arrives at a tablet server, the server checks that it is well-formed, and that the sender is authorized to perform the mutation. Authorization is performed by reading the list of permitted writers from a Chubby file (which is almost always a hit in the Chubby client cache). A valid mutation is written to the commit log. Group commit is used to improve the throughput of lots of small mutations [13, 16]. After the write has been committed, its contents are inserted into the memtable.

When a read operation arrives at a tablet server, it is similarly checked for well-formedness and proper authorization. A valid read operation is executed on a merged view of the sequence of SSTables and the memtable. Since the SSTables and the memtable are lexicographically sorted data structures, the merged view can be formed efficiently.

Incoming read and write operations can continue while tablets are split and merged.

5.4 Compactions

As write operations execute, the size of the memtable increases. When the memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to an SSTable and written to GFS. This *minor compaction* process has two goals: it shrinks the memory usage of the tablet server, and it reduces the amount of data that has to be read from the commit log during recovery if this server dies. Incoming read and write operations can continue while compactions occur.

Every minor compaction creates a new SSTable. If this behavior continued unchecked, read operations might need to merge updates from an arbitrary number of SSTables. Instead, we bound the number of such files by periodically executing a *merging compaction* in the background. A merging compaction reads the contents of a few SSTables and the memtable, and writes out a new SSTable. The input SSTables and memtable can be discarded as soon as the compaction has finished.

A merging compaction that rewrites all SSTables into exactly one SSTable is called a *major compaction*. SSTables produced by non-major compactions can contain special deletion entries that suppress deleted data in older SSTables that are still live. A major compaction, on the other hand, produces an SSTable that contains no deletion information or deleted data. Bigtable cycles through all of its tablets and regularly applies major compactions to them. These major compactions allow Bigtable to reclaim resources used by deleted data, and also allow it to ensure that deleted data disappears from the system in a timely fashion, which is important for services that store sensitive data.

6 Refinements

The implementation described in the previous section required a number of refinements to achieve the high performance, availability, and reliability required by our users. This section describes portions of the implementation in more detail in order to highlight these refinements.

Locality groups

Clients can group multiple column families together into a *locality group*. A separate SSTable is generated for each locality group in each tablet. Segregating column families that are not typically accessed together into separate locality groups enables more efficient reads. For example, page metadata in Webtable (such as language and checksums) can be in one locality group, and the contents of the page can be in a different group: an ap-

plication that wants to read the metadata does not need to read through all of the page contents.

In addition, some useful tuning parameters can be specified on a per-locality group basis. For example, a locality group can be declared to be in-memory. SSTables for in-memory locality groups are loaded lazily into the memory of the tablet server. Once loaded, column families that belong to such locality groups can be read without accessing the disk. This feature is useful for small pieces of data that are accessed frequently: we use it internally for the location column family in the METADATA table.

Compression

Clients can control whether or not the SSTables for a locality group are compressed, and if so, which compression format is used. The user-specified compression format is applied to each SSTable block (whose size is controllable via a locality group specific tuning parameter). Although we lose some space by compressing each block separately, we benefit in that small portions of an SSTable can be read without decompressing the entire file. Many clients use a two-pass custom compression scheme. The first pass uses Bentley and McIlroy's scheme [6], which compresses long common strings across a large window. The second pass uses a fast compression algorithm that looks for repetitions in a small 16 KB window of the data. Both compression passes are very fast-they encode at 100-200 MB/s, and decode at 400-1000 MB/s on modern machines.

Even though we emphasized speed instead of space reduction when choosing our compression algorithms, this two-pass compression scheme does surprisingly well. For example, in Webtable, we use this compression scheme to store Web page contents. In one experiment, we stored a large number of documents in a compressed locality group. For the purposes of the experiment, we limited ourselves to one version of each document instead of storing all versions available to us. The scheme achieved a 10-to-1 reduction in space. This is much better than typical Gzip reductions of 3-to-1 or 4-to-1 on HTML pages because of the way Webtable rows are laid out: all pages from a single host are stored close to each other. This allows the Bentley-McIlroy algorithm to identify large amounts of shared boilerplate in pages from the same host. Many applications, not just Webtable, choose their row names so that similar data ends up clustered, and therefore achieve very good compression ratios. Compression ratios get even better when we store multiple versions of the same value in Bigtable.

Caching for read performance

To improve read performance, tablet servers use two levels of caching. The Scan Cache is a higher-level cache that caches the key-value pairs returned by the SSTable interface to the tablet server code. The Block Cache is a lower-level cache that caches SSTables blocks that were read from GFS. The Scan Cache is most useful for applications that tend to read the same data repeatedly. The Block Cache is useful for applications that tend to read data that is close to the data they recently read (e.g., sequential reads, or random reads of different columns in the same locality group within a hot row).

Bloom filters

As described in Section 5.3, a read operation has to read from all SSTables that make up the state of a tablet. If these SSTables are not in memory, we may end up doing many disk accesses. We reduce the number of accesses by allowing clients to specify that Bloom filters [7] should be created for SSTables in a particular locality group. A Bloom filter allows us to ask whether an SSTable might contain any data for a specified row/column pair. For certain applications, a small amount of tablet server memory used for storing Bloom filters drastically reduces the number of disk seeks required for read operations. Our use of Bloom filters also implies that most lookups for non-existent rows or columns do not need to touch disk.

Commit-log implementation

If we kept the commit log for each tablet in a separate log file, a very large number of files would be written concurrently in GFS. Depending on the underlying file system implementation on each GFS server, these writes could cause a large number of disk seeks to write to the different physical log files. In addition, having separate log files per tablet also reduces the effectiveness of the group commit optimization, since groups would tend to be smaller. To fix these issues, we append mutations to a single commit log per tablet server, co-mingling mutations for different tablets in the same physical log file [18, 20].

Using one log provides significant performance benefits during normal operation, but it complicates recovery. When a tablet server dies, the tablets that it served will be moved to a large number of other tablet servers: each server typically loads a small number of the original server's tablets. To recover the state for a tablet, the new tablet server needs to reapply the mutations for that tablet from the commit log written by the original tablet server. However, the mutations for these tablets

were co-mingled in the same physical log file. One approach would be for each new tablet server to read this full commit log file and apply just the entries needed for the tablets it needs to recover. However, under such a scheme, if 100 machines were each assigned a single tablet from a failed tablet server, then the log file would be read 100 times (once by each server).

We avoid duplicating log reads by first sorting the commit log entries in order of the keys (table, row name, log sequence number). In the sorted output, all mutations for a particular tablet are contiguous and can therefore be read efficiently with one disk seek followed by a sequential read. To parallelize the sorting, we partition the log file into 64 MB segments, and sort each segment in parallel on different tablet servers. This sorting process is coordinated by the master and is initiated when a tablet server indicates that it needs to recover mutations from some commit log file.

Writing commit logs to GFS sometimes causes performance hiccups for a variety of reasons (e.g., a GFS server machine involved in the write crashes, or the network paths traversed to reach the particular set of three GFS servers is suffering network congestion, or is heavily loaded). To protect mutations from GFS latency spikes, each tablet server actually has two log writing threads, each writing to its own log file; only one of these two threads is actively in use at a time. If writes to the active log file are performing poorly, the log file writing is switched to the other thread, and mutations that are in the commit log queue are written by the newly active log writing thread. Log entries contain sequence numbers to allow the recovery process to elide duplicated entries resulting from this log switching process.

Speeding up tablet recovery

If the master moves a tablet from one tablet server to another, the source tablet server first does a minor compaction on that tablet. This compaction reduces recovery time by reducing the amount of uncompacted state in the tablet server's commit log. After finishing this compaction, the tablet server stops serving the tablet. Before it actually unloads the tablet, the tablet server does another (usually very fast) minor compaction to eliminate any remaining uncompacted state in the tablet server's log that arrived while the first minor compaction was being performed. After this second minor compaction is complete, the tablet can be loaded on another tablet server without requiring any recovery of log entries.

Exploiting immutability

Besides the SSTable caches, various other parts of the Bigtable system have been simplified by the fact that all of the SSTables that we generate are immutable. For example, we do not need any synchronization of accesses to the file system when reading from SSTables. As a result, concurrency control over rows can be implemented very efficiently. The only mutable data structure that is accessed by both reads and writes is the memtable. To reduce contention during reads of the memtable, we make each memtable row copy-on-write and allow reads and writes to proceed in parallel.

Since SSTables are immutable, the problem of permanently removing deleted data is transformed to garbage collecting obsolete SSTables. Each tablet's SSTables are registered in the METADATA table. The master removes obsolete SSTables as a mark-and-sweep garbage collection [25] over the set of SSTables, where the METADATA table contains the set of roots.

Finally, the immutability of SSTables enables us to split tablets quickly. Instead of generating a new set of SSTables for each child tablet, we let the child tablets share the SSTables of the parent tablet.

7 Performance Evaluation

We set up a Bigtable cluster with N tablet servers to measure the performance and scalability of Bigtable as N is varied. The tablet servers were configured to use 1 GB of memory and to write to a GFS cell consisting of 1786 machines with two 400 GB IDE hard drives each. N client machines generated the Bigtable load used for these tests. (We used the same number of clients as tablet servers to ensure that clients were never a bottleneck.) Each machine had two dual-core Opteron 2 GHz chips, enough physical memory to hold the working set of all running processes, and a single gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

The tablet servers and master, test clients, and GFS servers all ran on the same set of machines. Every machine ran a GFS server. Some of the machines also ran either a tablet server, or a client process, or processes from other jobs that were using the pool at the same time as these experiments.

R is the distinct number of Bigtable row keys involved in the test. R was chosen so that each benchmark read or wrote approximately 1 GB of data per tablet server.

The sequential write benchmark used row keys with names 0 to R-1. This space of row keys was partitioned into 10N equal-sized ranges. These ranges were assigned to the N clients by a central scheduler that as-

	# of Tablet Servers			
Experiment	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

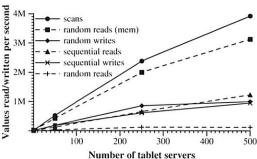


Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

signed the next available range to a client as soon as the client finished processing the previous range assigned to it. This dynamic assignment helped mitigate the effects of performance variations caused by other processes running on the client machines. We wrote a single string under each row key. Each string was generated randomly and was therefore uncompressible. In addition, strings under different row key were distinct, so no cross-row compression was possible. The *random write* benchmark was similar except that the row key was hashed modulo R immediately before writing so that the write load was spread roughly uniformly across the entire row space for the entire duration of the benchmark.

The *sequential read* benchmark generated row keys in exactly the same way as the sequential write benchmark, but instead of writing under the row key, it read the string stored under the row key (which was written by an earlier invocation of the sequential write benchmark). Similarly, the *random read* benchmark shadowed the operation of the random write benchmark.

The *scan* benchmark is similar to the sequential read benchmark, but uses support provided by the Bigtable API for scanning over all values in a row range. Using a scan reduces the number of RPCs executed by the benchmark since a single RPC fetches a large sequence of values from a tablet server.

The random reads (mem) benchmark is similar to the random read benchmark, but the locality group that contains the benchmark data is marked as *in-memory*, and therefore the reads are satisfied from the tablet server's memory instead of requiring a GFS read. For just this benchmark, we reduced the amount of data per tablet server from 1 GB to 100 MB so that it would fit comfortably in the memory available to the tablet server.

Figure 6 shows two views on the performance of our benchmarks when reading and writing 1000-byte values to Bigtable. The table shows the number of operations per second per tablet server; the graph shows the aggregate number of operations per second.

Single tablet-server performance

Let us first consider performance with just one tablet server. Random reads are slower than all other operations by an order of magnitude or more. Each random read involves the transfer of a 64 KB SSTable block over the network from GFS to a tablet server, out of which only a single 1000-byte value is used. The tablet server executes approximately 1200 reads per second, which translates into approximately 75 MB/s of data read from GFS. This bandwidth is enough to saturate the tablet server CPUs because of overheads in our networking stack, SSTable parsing, and Bigtable code, and is also almost enough to saturate the network links used in our system. Most Bigtable applications with this type of an access pattern reduce the block size to a smaller value, typically 8KB.

Random reads from memory are much faster since each 1000-byte read is satisfied from the tablet server's local memory without fetching a large 64 KB block from GFS.

Random and sequential writes perform better than random reads since each tablet server appends all incoming writes to a single commit log and uses group commit to stream these writes efficiently to GFS. There is no significant difference between the performance of random writes and sequential writes; in both cases, all writes to the tablet server are recorded in the same commit log.

Sequential reads perform better than random reads since every 64 KB SSTable block that is fetched from GFS is stored into our block cache, where it is used to serve the next 64 read requests.

Scans are even faster since the tablet server can return a large number of values in response to a single client RPC, and therefore RPC overhead is amortized over a large number of values.

Scaling

Aggregate throughput increases dramatically, by over a factor of a hundred, as we increase the number of tablet servers in the system from 1 to 500. For example, the

# of tablet servers		# of clusters	
0		19	259
20		49	47
50		99	20
100		499	50
> 500			12

Table 1: Distribution of number of tablet servers in Bigtable clusters.

performance of random reads from memory increases by almost a factor of 300 as the number of tablet server increases by a factor of 500. This behavior occurs because the bottleneck on performance for this benchmark is the individual tablet server CPU.

However, performance does not increase linearly. For most benchmarks, there is a significant drop in per-server throughput when going from 1 to 50 tablet servers. This drop is caused by imbalance in load in multiple server configurations, often due to other processes contending for CPU and network. Our load balancing algorithm attempts to deal with this imbalance, but cannot do a perfect job for two main reasons: rebalancing is throttled to reduce the number of tablet movements (a tablet is unavailable for a short time, typically less than one second, when it is moved), and the load generated by our benchmarks shifts around as the benchmark progresses.

The random read benchmark shows the worst scaling (an increase in aggregate throughput by only a factor of 100 for a 500-fold increase in number of servers). This behavior occurs because (as explained above) we transfer one large 64KB block over the network for every 1000-byte read. This transfer saturates various shared 1 Gigabit links in our network and as a result, the per-server throughput drops significantly as we increase the number of machines.

8 Real Applications

As of August 2006, there are 388 non-test Bigtable clusters running in various Google machine clusters, with a combined total of about 24,500 tablet servers. Table 1 shows a rough distribution of tablet servers per cluster. Many of these clusters are used for development purposes and therefore are idle for significant periods. One group of 14 busy clusters with 8069 total tablet servers saw an aggregate volume of more than 1.2 million requests per second, with incoming RPC traffic of about 741 MB/s and outgoing RPC traffic of about 16 GB/s.

Table 2 provides some data about a few of the tables currently in use. Some tables store data that is served to users, whereas others store data for batch processing; the tables range widely in total size, average cell size,

percentage of data served from memory, and complexity of the table schema. In the rest of this section, we briefly describe how three product teams use Bigtable.

8.1 Google Analytics

Google Analytics (analytics.google.com) is a service that helps webmasters analyze traffic patterns at their web sites. It provides aggregate statistics, such as the number of unique visitors per day and the page views per URL per day, as well as site-tracking reports, such as the percentage of users that made a purchase, given that they earlier viewed a specific page.

To enable the service, webmasters embed a small JavaScript program in their web pages. This program is invoked whenever a page is visited. It records various information about the request in Google Analytics, such as a user identifier and information about the page being fetched. Google Analytics summarizes this data and makes it available to webmasters.

We briefly describe two of the tables used by Google Analytics. The raw click table (~200 TB) maintains a row for each end-user session. The row name is a tuple containing the website's name and the time at which the session was created. This schema ensures that sessions that visit the same web site are contiguous, and that they are sorted chronologically. This table compresses to 14% of its original size.

The summary table (~20 TB) contains various predefined summaries for each website. This table is generated from the raw click table by periodically scheduled MapReduce jobs. Each MapReduce job extracts recent session data from the raw click table. The overall system's throughput is limited by the throughput of GFS. This table compresses to 29% of its original size.

8.2 Google Earth

Google operates a collection of services that provide users with access to high-resolution satellite imagery of the world's surface, both through the web-based Google Maps interface (maps.google.com) and through the Google Earth (earth.google.com) custom client software. These products allow users to navigate across the world's surface: they can pan, view, and annotate satellite imagery at many different levels of resolution. This system uses one table to preprocess data, and a different set of tables for serving client data.

The preprocessing pipeline uses one table to store raw imagery. During preprocessing, the imagery is cleaned and consolidated into final serving data. This table contains approximately 70 terabytes of data and therefore is served from disk. The images are efficiently compressed already, so Bigtable compression is disabled.

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency- sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	-	9	8	3	0%	No
Orkut	9	_	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and # *Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.

Each row in the imagery table corresponds to a single geographic segment. Rows are named to ensure that adjacent geographic segments are stored near each other. The table contains a column family to keep track of the sources of data for each segment. This column family has a large number of columns: essentially one for each raw data image. Since each segment is only built from a few images, this column family is very sparse.

The preprocessing pipeline relies heavily on MapReduce over Bigtable to transform data. The overall system processes over 1 MB/sec of data per tablet server during some of these MapReduce jobs.

The serving system uses one table to index data stored in GFS. This table is relatively small (~500 GB), but it must serve tens of thousands of queries per second per datacenter with low latency. As a result, this table is hosted across hundreds of tablet servers and contains inmemory column families.

8.3 Personalized Search

Personalized Search (www.google.com/psearch) is an opt-in service that records user queries and clicks across a variety of Google properties such as web search, images, and news. Users can browse their search histories to revisit their old queries and clicks, and they can ask for personalized search results based on their historical Google usage patterns.

Personalized Search stores each user's data in Bigtable. Each user has a unique userid and is assigned a row named by that userid. All user actions are stored in a table. A separate column family is reserved for each type of action (for example, there is a column family that stores all web queries). Each data element uses as its Bigtable timestamp the time at which the corresponding user action occurred. Personalized Search generates user profiles using a MapReduce over Bigtable. These user profiles are used to personalize live search results.

The Personalized Search data is replicated across several Bigtable clusters to increase availability and to reduce latency due to distance from clients. The Personalized Search team originally built a client-side replication mechanism on top of Bigtable that ensured eventual consistency of all replicas. The current system now uses a replication subsystem that is built into the servers.

The design of the Personalized Search storage system allows other groups to add new per-user information in their own columns, and the system is now used by many other Google properties that need to store per-user configuration options and settings. Sharing a table amongst many groups resulted in an unusually large number of column families. To help support sharing, we added a simple quota mechanism to Bigtable to limit the storage consumption by any particular client in shared tables; this mechanism provides some isolation between the various product groups using this system for per-user information storage.

9 Lessons

In the process of designing, implementing, maintaining, and supporting Bigtable, we gained useful experience and learned several interesting lessons.

One lesson we learned is that large distributed systems are vulnerable to many types of failures, not just the standard network partitions and fail-stop failures assumed in many distributed protocols. For example, we have seen problems due to all of the following causes: memory and network corruption, large clock skew, hung machines, extended and asymmetric network partitions, bugs in other systems that we are using (Chubby for example), overflow of GFS quotas, and planned and unplanned hardware maintenance. As we have gained more experience with these problems, we have addressed them by changing various protocols. For example, we added checksumming to our RPC mechanism. We also handled

some problems by removing assumptions made by one part of the system about another part. For example, we stopped assuming a given Chubby operation could return only one of a fixed set of errors.

Another lesson we learned is that it is important to delay adding new features until it is clear how the new features will be used. For example, we initially planned to support general-purpose transactions in our API. Because we did not have an immediate use for them, however, we did not implement them. Now that we have many real applications running on Bigtable, we have been able to examine their actual needs, and have discovered that most applications require only single-row transactions. Where people have requested distributed transactions, the most important use is for maintaining secondary indices, and we plan to add a specialized mechanism to satisfy this need. The new mechanism will be less general than distributed transactions, but will be more efficient (especially for updates that span hundreds of rows or more) and will also interact better with our scheme for optimistic cross-data-center replication.

A practical lesson that we learned from supporting Bigtable is the importance of proper system-level monitoring (i.e., monitoring both Bigtable itself, as well as the client processes using Bigtable). For example, we extended our RPC system so that for a sample of the RPCs, it keeps a detailed trace of the important actions done on behalf of that RPC. This feature has allowed us to detect and fix many problems such as lock contention on tablet data structures, slow writes to GFS while committing Bigtable mutations, and stuck accesses to the METADATA table when METADATA tablets are unavailable. Another example of useful monitoring is that every Bigtable cluster is registered in Chubby. This allows us to track down all clusters, discover how big they are, see which versions of our software they are running, how much traffic they are receiving, and whether or not there are any problems such as unexpectedly large latencies.

The most important lesson we learned is the value of simple designs. Given both the size of our system (about 100,000 lines of non-test code), as well as the fact that code evolves over time in unexpected ways, we have found that code and design clarity are of immense help in code maintenance and debugging. One example of this is our tablet-server membership protocol. Our first protocol was simple: the master periodically issued leases to tablet servers, and tablet servers killed themselves if their lease expired. Unfortunately, this protocol reduced availability significantly in the presence of network problems, and was also sensitive to master recovery time. We redesigned the protocol several times until we had a protocol that performed well. However, the resulting protocol was too complex and depended on

the behavior of Chubby features that were seldom exercised by other applications. We discovered that we were spending an inordinate amount of time debugging obscure corner cases, not only in Bigtable code, but also in Chubby code. Eventually, we scrapped this protocol and moved to a newer simpler protocol that depends solely on widely-used Chubby features.

10 Related Work

The Boxwood project [24] has components that overlap in some ways with Chubby, GFS, and Bigtable, since it provides for distributed agreement, locking, distributed chunk storage, and distributed B-tree storage. In each case where there is overlap, it appears that the Boxwood's component is targeted at a somewhat lower level than the corresponding Google service. The Boxwood project's goal is to provide infrastructure for building higher-level services such as file systems or databases, while the goal of Bigtable is to directly support client applications that wish to store data.

Many recent projects have tackled the problem of providing distributed storage or higher-level services over wide area networks, often at "Internet scale." This includes work on distributed hash tables that began with projects such as CAN [29], Chord [32], Tapestry [37], and Pastry [30]. These systems address concerns that do not arise for Bigtable, such as highly variable bandwidth, untrusted participants, or frequent reconfiguration; decentralized control and Byzantine fault tolerance are not Bigtable goals.

In terms of the distributed data storage model that one might provide to application developers, we believe the key-value pair model provided by distributed B-trees or distributed hash tables is too limiting. Key-value pairs are a useful building block, but they should not be the only building block one provides to developers. The model we chose is richer than simple key-value pairs, and supports sparse semi-structured data. Nonetheless, it is still simple enough that it lends itself to a very efficient flat-file representation, and it is transparent enough (via locality groups) to allow our users to tune important behaviors of the system.

Several database vendors have developed parallel databases that can store large volumes of data. Oracle's Real Application Cluster database [27] uses shared disks to store data (Bigtable uses GFS) and a distributed lock manager (Bigtable uses Chubby). IBM's DB2 Parallel Edition [4] is based on a shared-nothing [33] architecture similar to Bigtable. Each DB2 server is responsible for a subset of the rows in a table which it stores in a local relational database. Both products provide a complete relational model with transactions.

Bigtable locality groups realize similar compression and disk read performance benefits observed for other systems that organize data on disk using column-based rather than row-based storage, including C-Store [1, 34] and commercial products such as Sybase IQ [15, 36], SenSage [31], KDB+ [22], and the ColumnBM storage layer in MonetDB/X100 [38]. Another system that does vertical and horizontal data partioning into flat files and achieves good data compression ratios is AT&T's Daytona database [19]. Locality groups do not support CPUcache-level optimizations, such as those described by Ailamaki [2].

The manner in which Bigtable uses memtables and SSTables to store updates to tablets is analogous to the way that the Log-Structured Merge Tree [26] stores updates to index data. In both systems, sorted data is buffered in memory before being written to disk, and reads must merge data from memory and disk.

C-Store and Bigtable share many characteristics: both systems use a shared-nothing architecture and have two different data structures, one for recent writes, and one for storing long-lived data, with a mechanism for moving data from one form to the other. The systems differ significantly in their API: C-Store behaves like a relational database, whereas Bigtable provides a lower level read and write interface and is designed to support many thousands of such operations per second per server. C-Store is also a "read-optimized relational DBMS", whereas Bigtable provides good performance on both read-intensive and write-intensive applications.

Bigtable's load balancer has to solve some of the same kinds of load and memory balancing problems faced by shared-nothing databases (e.g., [11, 35]). Our problem is somewhat simpler: (1) we do not consider the possibility of multiple copies of the same data, possibly in alternate forms due to views or indices; (2) we let the user tell us what data belongs in memory and what data should stay on disk, rather than trying to determine this dynamically; (3) we have no complex queries to execute or optimize.

11 Conclusions

We have described Bigtable, a distributed system for storing structured data at Google. Bigtable clusters have been in production use since April 2005, and we spent roughly seven person-years on design and implementation before that date. As of August 2006, more than sixty projects are using Bigtable. Our users like the performance and high availability provided by the Bigtable implementation, and that they can scale the capacity of their clusters by simply adding more machines to the system as their resource demands change over time.

Given the unusual interface to Bigtable, an interesting question is how difficult it has been for our users to adapt to using it. New users are sometimes uncertain of how to best use the Bigtable interface, particularly if they are accustomed to using relational databases that support general-purpose transactions. Nevertheless, the fact that many Google products successfully use Bigtable demonstrates that our design works well in practice.

We are in the process of implementing several additional Bigtable features, such as support for secondary indices and infrastructure for building cross-data-center replicated Bigtables with multiple master replicas. We have also begun deploying Bigtable as a service to product groups, so that individual groups do not need to maintain their own clusters. As our service clusters scale, we will need to deal with more resource-sharing issues within Bigtable itself [3, 5].

Finally, we have found that there are significant advantages to building our own storage solution at Google. We have gotten a substantial amount of flexibility from designing our own data model for Bigtable. In addition, our control over Bigtable's implementation, and the other Google infrastructure upon which Bigtable depends, means that we can remove bottlenecks and inefficiencies as they arise.

Acknowledgements

We thank the anonymous reviewers, David Nagle, and our shepherd Brad Calder, for their feedback on this paper. The Bigtable system has benefited greatly from the feedback of our many users within Google. In addition, we thank the following people for their contributions to Bigtable: Dan Aguayo, Sameer Ajmani, Zhifeng Chen, Bill Coughran, Mike Epstein, Healfdene Goguen, Robert Griesemer, Jeremy Hylton, Josh Hyman, Alex Khesin, Joanna Kulik, Alberto Lerner, Sherry Listgarten, Mike Maloney, Eduardo Pinheiro, Kathy Polizzi, Frank Yellin, and Arthur Zwiegincew.

References

- ABADI, D. J., MADDEN, S. R., AND FERREIRA, M. C. Integrating compression and execution in columnoriented database systems. *Proc. of SIGMOD* (2006).
- [2] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SK-OUNAKIS, M. Weaving relations for cache performance. In *The VLDB Journal* (2001), pp. 169–180.
- [3] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd OSDI* (Feb. 1999), pp. 45–58.
- [4] BARU, C. K., FECTEAU, G., GOYAL, A., HSIAO, H., JHINGRAN, A., PADMANABHAN, S., COPELAND,

- G. P., AND WILSON, W. G. DB2 parallel edition. *IBM Systems Journal* 34, 2 (1995), 292–322.
- [5] BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating system support for planetary-scale network services. In *Proc. of the 1st NSDI* (Mar. 2004), pp. 253–266.
- [6] BENTLEY, J. L., AND MCILROY, M. D. Data compression using long common strings. In *Data Compression Conference* (1999), pp. 287–295.
- [7] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. CACM 13, 7 (1970), 422–426.
- [8] BURROWS, M. The Chubby lock service for looselycoupled distributed systems. In *Proc. of the 7th OSDI* (Nov. 2006).
- [9] CHANDRA, T., GRIESEMER, R., AND REDSTONE, J. Paxos made live — An engineering perspective. In *Proc.* of *PODC* (2007).
- [10] COMER, D. Ubiquitous B-tree. Computing Surveys 11, 2 (June 1979), 121–137.
- [11] COPELAND, G. P., ALEXANDER, W., BOUGHTER, E. E., AND KELLER, T. W. Data placement in Bubba. In *Proc. of SIGMOD* (1988), pp. 99–108.
- [12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th OSDI* (Dec. 2004), pp. 137–150.
- [13] DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. Implementation techniques for main memory database systems. In *Proc.* of SIGMOD (June 1984), pp. 1–8.
- [14] DEWITT, D. J., AND GRAY, J. Parallel database systems: The future of high performance database systems. *CACM* 35, 6 (June 1992), 85–98.
- [15] FRENCH, C. D. One size fits all database architectures do not work for DSS. In *Proc. of SIGMOD* (May 1995), pp. 449–450.
- [16] GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS fast path. *Database Engineering Bulletin* 8, 2 (1985), 3–10.
- [17] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proc. of the 19th ACM SOSP* (Dec. 2003), pp. 29–43.
- [18] GRAY, J. Notes on database operating systems. In Operating Systems An Advanced Course, vol. 60 of Lecture Notes in Computer Science. Springer-Verlag, 1978.
- [19] GREER, R. Daytona and the fourth-generation language Cymbal. In *Proc. of SIGMOD* (1999), pp. 525–526.
- [20] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proc. of the 11th* SOSP (Dec. 1987), pp. 155–162.
- [21] HARTMAN, J. H., AND OUSTERHOUT, J. K. The Zebra striped network file system. In *Proc. of the 14th SOSP* (Asheville, NC, 1993), pp. 29–43.

- [22] KX.COM. kx.com/products/database.php. Product page.
- [23] LAMPORT, L. The part-time parliament. ACM TOCS 16, 2 (1998), 133–169.
- [24] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc.* of the 6th OSDI (Dec. 2004), pp. 105–120.
- [25] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. CACM 3, 4 (Apr. 1960), 184–195.
- [26] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). Acta Inf. 33, 4 (1996), 351–385.
- [27] ORACLE.COM. www.oracle.com/technology/products/database/clustering/index.html. Product page.
- [28] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUIN-LAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal* 13, 4 (2005), 227–298.
- [29] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. of SIGCOMM* (Aug. 2001), pp. 161– 172.
- [30] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for largescale peer-to-peer systems. In *Proc. of Middleware 2001* (Nov. 2001), pp. 329–350.
- [31] SENSAGE.COM. sensage.com/products-sensage.htm.
 Product page.
- [32] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of SIGCOMM* (Aug. 2001), pp. 149–160.
- [33] STONEBRAKER, M. The case for shared nothing. Database Engineering Bulletin 9, 1 (Mar. 1986), 4–9.
- [34] STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. C-Store: A columnoriented DBMS. In *Proc. of VLDB* (Aug. 2005), pp. 553– 564.
- [35] STONEBRAKER, M., AOKI, P. M., DEVINE, R., LITWIN, W., AND OLSON, M. A. Mariposa: A new architecture for distributed data. In *Proc. of the Tenth ICDE* (1994), IEEE Computer Society, pp. 54–65.
- [36] SYBASE.COM. www.sybase.com/products/databaseservers/sybaseiq. Product page.
- [37] ZHAO, B. Y., KUBIATOWICZ, J., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, CS Division, UC Berkeley, Apr. 2001.
- [38] ZUKOWSKI, M., BONCZ, P. A., NES, N., AND HEMAN, S. MonetDB/X100 A DBMS in the CPU cache. *IEEE Data Eng. Bull.* 28, 2 (2005), 17–22.

EnsemBlue: Integrating Distributed Storage and Consumer Electronics

Daniel Peek and Jason Flinn
Department of Electrical Engineering and Computer Science
University of Michigan

Abstract

EnsemBlue is a distributed file system for personal multimedia that incorporates both general-purpose computers and consumer electronic devices (CEDs). EnsemBlue leverages the capabilities of a few general-purpose computers to make CEDs first class clients of the file system. It supports namespace diversity by translating between its distributed namespace and the local namespaces of CEDs. It supports extensibility through persistent queries, a robust event notification mechanism that leverages the underlying cache consistency protocols of the file system. Finally, it allows mobile clients to self-organize and share data through device ensembles. Our results show that these features impose little overhead, yet they enable the integration of emerging platforms such as digital cameras, MP3 players, and DVRs.

1 Introduction

Consumer electronic devices (CEDs) are increasingly important computing platforms. CEDs differ from general-purpose computers in both their degree of specialization and the narrowness of their interfaces. As predicted by Weiser [27], these computers "disappear into the background" because they present a specialized interface that is limited to the particular application for which they are designed. Nevertheless, CEDs are often formidable computing platforms that possess substantial storage, processing, and networking capabilities.

In this paper, we explore how CEDs can be integrated into a distributed file system. Our focus on storage is motivated by the difficulty of managing personal multimedia such as photos, video, and music. Current approaches to organizing data (e.g., manual synchronization) do not scale well as the number of computers and CEDs owned by a single user increases. For instance, when files are manually associated with specific devices, users must intervene to decide which items are replicated on which device. Users must also manage consistency of replicated data when files are updated. To guard against data loss, users must place copies of data in reliable, well-maintained storage locations.

Since distributed file systems have successfully automated these time-consuming and error-prone tasks in workstation environments [3, 7], we posit that they can perform a similar function for the home user. To explore our hypothesis, we have created EnsemBlue, a distributed file system that is designed to store personal multimedia. EnsemBlue, which is based on the Blue File System [14], adds several new capabilities to support consumer electronic devices:

- persistent queries. The heterogeneity of CEDs creates the need to customize file system behavior. For instance, many CEDs are associated with files of only one type; e.g., a digital camera with JPEGs. The network, storage, and processing resources of different CEDs vary widely since each is equipped with only the resources required to perform its particular function. Rather than treat each device equally, Ensem-Blue provides persistent queries that customize its behavior for each client. A persistent query delivers event notifications to applications that specialize file system behavior. Notifications can be delivered to applications running on any EnsemBlue client; they are robust in the presence of failures and network disconnection. Persistent queries have low overhead because they reuse the existing cache consistency protocols of the file system to deliver notifications. They serve as the foundation on which to build custom automation such as type-specific caching, transcoding, and application-specific indexing.
- namespace diversity. Many CEDs have custom organizations for the data they store (e.g., an iPod stores music files in several different subdirectories in its local file system). The organization of data on a CED is application-specific; it is not necessarily the way the user most naturally thinks about the data. Since no single namespace can suffice for all CEDs and general-purpose computers, EnsemBlue supports namespace diversity. It creates a distributed namespace for a user's personal data that is shared among general-purpose computers the user can organize this namespace in any fashion. It supports CEDs with

custom file organizations by automatically translating between the distributed namespace and each device-specific namespace. Changes made in the EnsemBlue namespace trigger equivalent changes in CED namespaces. Modifications made within a CED namespace are automatically propagated to the EnsemBlue namespace and shared with other clients.

• ensemble support. A mobile user may carry several CEDs; e.g., a cell phone, an MP3 player, and a portable DVD player. Device ensembles (sometimes called personal area networks) let multiple mobile computers and CEDs owned by the same user self-organize and share data [21]. For instance, an MP3 player might play not just music stored on its local hard drive, but also music stored on a co-located cell phone or laptop. EnsemBlue allows its mobile clients to form ensembles and directly access data from other clients. It presents a consistent view of data within each ensemble by propagating changes made on one device to replicas on other ensemble members.

Our results show that the overhead of these new capabilities is minimal. We present case studies in which we use these capabilities to integrate a digital camera and an iPod with EnsemBlue. We demonstrate that the extensibility of EnsemBlue enables a high degree of automation, including the ability to automatically organize photos and music, index text and multimedia data, and transcode content to support different media players.

2 Background

When we began this work, our ambition was to develop a distributed file system to store personal multimedia. We focused on multimedia because of the explosion of computing and consumer electronics devices that consume that type of data. Anecdotally, we found that many of our acquaintances owned several devices and that managing personal multimedia was increasingly a chore.

We adapted BlueFS [14], a distributed file system developed by our research group, to accomplish this task. Our choice of BlueFS was driven by several factors. BlueFS allows clients to operate disconnected, a critical ability for devices such as MP3 players and laptops that often operate without a network connection. BlueFS is also designed to conserve the battery lifetime of mobile clients. BlueFS has first class support for portable storage, which we felt would be useful for devices such as iPods and cameras. Finally, BlueFS lets clients dynamically add and remove storage devices — this seemed to mesh well with users who occasionally synchronize their CEDs with a general-purpose computer.

Over the past five months, our research group has used BlueFS to store personal multimedia. Our initial experience has been encouraging in several respects. We have found the common namespace of a distributed file system to be a useful way to organize data. One member of the group uses BlueFS to play music on a home PC, a laptop, a work computer, a TiVo DVR in his living room, and a D-Link wireless media player. After adding a new song to his BlueFS volume on any computer, he can play it from any of these locations. We have used support for disconnected operation to display content on long plane flights. The abilities of BlueFS to cache files on local disk and reintegrate modifications asynchronously have also proven useful. The group BlueFS file server is located at the University of Michigan, while the majority of clients are in home locations connected via broadband links. Disk caching reduces the frequency of skips when listening to music at home because it avoids a potentially unreliable broadband link. Further, when storing large video files that have been recorded at home, the ability to reintegrate modifications asynchronously has helped hide network constraints.

Unfortunately, our initial experience storing personal multimedia in BlueFS also revealed several problems. Most troubling was our inability to use BlueFS with many of our favorite CEDs such as cameras and MP3 players. Because these devices are closed platforms, they could not run the BlueFS client code. These devices required specific file organizations that did not match the way we had organized our files in the distributed namespace. The CEDs with which we had the most success were ones such as the TiVo DVR that use third-party software to interface with the local file system of a home computer. If the home computer exports the BlueFS namespace, such CEDs can indirectly read from and write to BlueFS files.

We found that the mechanisms for managing caching in BlueFS were insufficiently expressive. These mechanisms let us control caching according to the location of files within the hierarchical directory structure. However, we often wanted richer semantics. For instance, we wanted to cache all files of a certain type on particular devices (e.g., all MP3s on a laptop). Since large files were particularly time-consuming to transfer between home and work over a broadband connection, we wanted to specify policies where large files would be cached in both locations. Unfortunately, limited caching semantics constrained how we organized files. For example, to control the caching of JPEG photos, we put all files of that type in a single file system subtree.

We wished that BlueFS were more extensible. Since several of our media players supported a limited set of formats, we found it necessary to transcode files. As transcoding is CPU intensive, we used workstations for this task, which required us to log in to these machines remotely. Instead of performing this task manually each time new media files were added to the file system, we would have preferred to extend the file system to do transcoding automatically.

Finally, we were sometimes frustrated by the need to propagate updates between clients through the file server. When both a producer and consumer of data were located at home, the broadband link connecting the home computers with the file server was a communication bottleneck. For instance, it would take many hours to propagate a video recorded on a DVR to a laptop because data had to traverse the bottleneck link twice. While we appreciated the file server as a safe repository for our data that was regularly backed up, we also wanted the ability to ship data between clients directly. We also believed that as we came to use more mobile devices, it would be useful to exchange data directly between them when they were disconnected from the server.

To address these problems, we have created a new file system, called EnsemBlue, that is based on the original BlueFS code base. EnsemBlue provides three novel capabilities, which we will describe in Sections 4–6: persistent queries to support customization of file system behavior, explicit support for closed-platform CEDs which require particular file system organizations, and ensemble support that allows multiple clients to exchange data without communicating with the file server.

3 Target environment

Individual EnsemBlue deployments are targeted at meeting the storage needs of a single user or a small group of users such as a family. A well-maintained file server might reside at home or with an ISP; its clients could include desktops, laptops, MP3 players, cell phones, and digital cameras. As many clients are mobile, EnsemBlue supports disconnected operation for isolated devices and allows collections of disconnected devices to form ensembles. Read-only sharing of content among different servers is enabled through a loosely-coupled federation mechanism.

Since EnsemBlue targets multimedia data, we expect that most files stored in the system will be large, and that reads will dominate writes. Updates, when they occur, will typically be small changes to file metadata such as song ratings and photo captions. EnsemBlue's consistency model is designed for a read-mostly workload. It uses a callback-based cache coherence strategy in which a client sets a callback with the server when it reads an object. The callback is a promise by the server to notify the client when the object is modified. Similar to Coda's weakly connected mode of operation, updates are propagated asynchronously to the file server. As in any system

that uses optimistic concurrency, conflicts can occur if two updates are made concurrently; if this occurs, EnsemBlue supports Coda-style conflict resolution.

4 Persistent queries

Persistent queries are a robust event notification mechanism that lets users customize the file system with applications that automate common tasks. With persistent queries, one can tune EnsemBlue's replication strategy to meet the needs of different CEDs. Persistent queries also let applications index or transcode files created by other computers and CEDs, even if those clients were disconnected from the network when files were added.

4.1 Design considerations

The first design issue we considered was how tightly to integrate custom functionality with EnsemBlue. We initially considered a tight integration that would allow custom code to be directly injected into the file system. However, we felt this approach would require careful sandboxing to address reliability, security, and privacy concerns. Therefore, we opted for a simpler, more loosely-coupled approach. We observed that, for local file systems, custom functionality is often implemented by standalone applications like the Glimpse indexer [13] or the lame transcoder [11]. However, existing distributed file systems do not provide a way for applications to learn about events that happen on other clients. Our approach, therefore, was to broaden the interface of EnsemBlue to better support standalone applications that extend file system behavior.

The functionality most sorely lacking was a robust event notification mechanism that supports multiple clients, some of which are mobile. Although current operating systems can notify applications about local file system changes [24], their notification mechanisms do not scale to distributed environments. For instance, a transcoder running on a laptop should be notified when JPEG files are added by other file system clients. The laptop may frequently disconnect from the network, complicating the delivery of notifications. Further, if JPEG files are added by a digital camera, the laptop and camera may rarely be connected to the network at the same time.

Potentially, we could have implemented a separate event notification framework. However, distributed file systems already provide notifications when files are modified in order to maintain cache consistency on multiple clients. Thus, by expressing event notifications as modifications to objects within the distributed file system, we can reuse the existing cache coherency mechanism of the distributed file system to deliver those notifications.

pq_create	(IN String query, IN Long event_mask, OUT Id fileid);	Creates a query and returns its unique identifier
pq_delete	(IN Id fileid);	Deletes the specified query
pq_open	(IN Id fileid, OUT Int file_descriptor);	Opens an existing query
pq_close	(IN Int file_descriptor);	Closes the specified query
pq_wait	(IN Int file_descriptor, IN Timeval tv);	Blocks until a record is available to read
pq_next	(IN Int file_descriptor, OUT event_record);	Returns the next record in the query log (if any)
$pq_truncate$	(IN Int file_descriptor, IN Int record_number);	Deletes records up to the specified record

Figure 1. Persistent query interface

A persistent query is a new type of file system object that is used to deliver event notifications. An application creates a persistent query to express the set of events that it is interested in receiving. The file server appends log records to the query when an event matching the query occurs. An application extending file system behavior reads the records from the query object, processes them, then removes them from the object. Since the persistent query is an object within the file system, the existing cache consistency mechanisms of EnsemBlue automatically propagate updates made by the server or application to the other party. EnsemBlue inherits the callbackbased cache consistency of BlueFS [14], which ensures that updates made by a disconnected client are propagated to the server when the client reconnects. Similarly, invalidations queued by the server while the client was disconnected are delivered when it reconnects.

For example, an application that transcodes M4A music to the MP3 format creates a persistent query so that it is informed when new M4A files are added. It opens the query and selects on the file descriptor to block until new events arrive. The EnsemBlue client sets a callback with the file server for the query (if one does not already exist) when the query is opened. If another client adds a new M4A file, the EnsemBlue server appends an event record to the query, which causes an invalidation to be sent to the client running the transcoder. That client refetches the query and unblocks the transcoder. After reading the event record, the transcoder creates the corresponding MP3 file.

We next considered the semantics for event notification. Given our decision to implement custom functionality in standalone applications, semantics that deliver each event exactly once did not seem appropriate. Events could be lost if an application or operating system crash occurs after a notification is delivered but before the event is processed. While we could potentially perform event notification and processing as an atomic transaction, this would necessitate a much tighter coupling of the file system and applications than we want.

Instead, we observed that customizations can usually be structured as idempotent operations. For instance, an indexing application can insert a newly created file into its index only if and only if it is not already present. Therefore, EnsemBlue provides *at least once* semantics

for event notification. A customization application first receives a notification, then processes it, and finally removes the event from the query. If a crash occurs before the application processes the event, the notification is preserved since the query is a persistent object. If a crash occurs after the application processes the event but before it removes it from the query, it will reread the same notification on restart. Since its event processing is idempotent, the result is the same as if it had received only one notification.

4.2 Implementation

Figure 1 shows the interface for persistent queries. Applications running on any EnsemBlue client can create a new query by calling pq_create and specifying both a query string expressed over file metadata and an event mask that specifies the set of file system events on which the query string should be evaluated. Currently, the query string can be expressed over a subset of metadata fields (e.g., file name, owner, etc.). The event mask contains a bit for each modification type; e.g., it has bits for file creation and deletion.

Like directories, queries are a separate type of file system object that have a restricted application interface. A query contains both header information (the query string and event mask) and a log of events that match the query. Each record contains the event type as well as the 96 bit EnsemBlue unique identifier for the matching file.

The server keeps a list of outstanding queries. When it processes a modification, it checks all queries for the modification type to see if the state of any modified object matches any query string. If a match occurs, the server appends an event record to the query. The server guarantees that the appending of any event record is atomic with the modification by committing both updates to disk in the same transaction. Since queries are persistent, if an update is made to the file system, matching event notifications are eventually delivered.

An event mask also contains an *initial* bit that applications set to evaluate a query over the current state of the file system. If this bit is set, the server adds a record to the query for each existing file that matches the query string. If an application sets both the initial and file creation bits, the server guarantees that the application is no-

tified about all files that match the query string, including those created concurrently with the creation of the query.

Several implementation choices improve response time when evaluating persistent queries. First, queries are evaluated at the server, which typically has better computational resources than CEDs and other clients. Evaluating queries at the server also benefits from moving computation close to the data since the server stores the primary replica of every object. In contrast, evaluating queries at a client would require the client to fetch data from the server for each uncached object. Second, the server maintains an in-memory index of file metadata that it uses to answer queries over existing file system state. Use of an index means that the server does not need to scan all objects that it stores to answer each query. Finally, after the query is initially created, all further evaluation is incremental. Because the server makes each new record persistent atomically with the operation that caused the record to be written, query records are not lost due to crash or power failure. This eliminates the need to rescan the entire file system on recovery.

A drawback of making queries persistent is that queries that are no longer useful may accumulate over time. We plan to address this with a tool that periodically examines the outstanding queries and deletes ones that have not been used for a substantial period of time. Another potential drawback is that updating persistent queries creates additional serialization delays due to lock acquisition; however, since locks are held only briefly, the serialization costs in the server are usually negligible when compared with disk I/O costs. Since a query update is committed in the same disk transaction as the file operation that caused the update, the query update does not require extra disk seeks.

4.3 Examples

We have built four examples of customized functionality that use persistent queries. The first is a multimedia transcoder that converts M4A music to the MP3 format. When the transcoder first runs, it creates a query that matches existing files that have a name ending in ".m4a", as well as update and creation events for files of that name. As with many current multimedia programs, applications built on persistent queries typically infer a file's type from its name. All such applications share an implicit assumption that common naming conventions are employed while creating files.

When the transcoder is notified of a new M4A file, it invokes the Linux faad and lame tools to convert between the two formats. It stores the resulting MP3 in the same directory as the original M4A file. Since persistent queries deliver notifications to any EnsemBlue

client, we run the transcoder on a PC with ample computational resources. If an M4A file were to be added by a disconnected CED such as a cell phone, the notification reaches the transcoder once the phone reconnects to the file server. This transcoder is only 108 lines of code.

We also use persistent queries to support *type-specific affinity*. A command line tool lets users specify caching behavior for any storage device as a query string. The tool sets the initial bit in the event mask, as well as the bits for events that create new files, delete files, or modify them. When an event for an existing or newly created file is added to the query, the file is fetched and cached on the storage device. For example, using type-specific affinity, one can cache all music files on an MP3 player to allow them to be played while disconnected, or one can cache large files on a home PC to avoid communication delays associated with a broadband link.

The last two examples of persistent queries perform typespecific indexing. Applications such as iTunes, Spotlight, and Glimpse are examples of popular tools that index data stored on a local file system. However, because these tools rely on event notification mechanisms that are confined to a single computer, they do not scale well to distributed file systems. We augmented two existing tools, Glimpse and GnuPod, to use persistent queries. The Glimpse indexer creates a query that matches all events (creation, deletion, update, etc.) for files that contain textual data. The music indexer matches the same events for MP3 and other music files. The first time these tools execute, they index the files currently in a user's EnsemBlue namespace. Afterward, they incrementally update their databases when they are notified of the addition or deletion of matching files. The indexers run on powerful machines with spare cycles to reduce latency, yet their results are accessible from any client because they are stored in EnsemBlue. We used 139 lines of code to add persistent query support to Glimpse and 86 lines of code for GnuPod.

5 Integrating consumer electronic devices

5.1 Leveraging general-purpose computers

At first glance, it appears that closed-platform CEDs cannot participate in a distributed file system because they lack the extensibility to execute custom file system code. For instance, most DVRs and MP3 players require substantial hacking to modify them to run arbitrary executables. Even CEDs that are extensible at the user level may not allow kernel modifications.

EnsemBlue circumvents this problem by leveraging the capabilities of general-purpose computers. A closed-platform CED can participate in EnsemBlue by *attaching* to any general-purpose client of its file server. The

EnsemBlue daemon, Wolverine, running on the generalpurpose client acts on behalf of the CED for all Ensem-Blue activities. If the user modifies data in the local CED namespace, Wolverine detects these changes and makes corresponding updates to the distributed Ensem-Blue namespace. Similarly, when data is modified in the EnsemBlue namespace, Wolverine propagates relevant modifications to the CED.

The requirements for a closed-platform CED to participate in EnsemBlue are minimal. The CED must support communication with a general-purpose computer; e.g., via a wireless interface or USB cable. The CED must provide a method to list the files it stores, as well as methods to read and update each file. Currently, EnsemBlue supports CEDs that provide a file system interface such as FAT and cameras that support the Picture Transfer Protocol.

5.2 Making CEDs self-describing

In contrast to current models that require CEDs to synchronize with particular computers, EnsemBlue allows CEDs to attach to any general-purpose client, even if that client is currently disconnected from the file server. In order to provide this flexibility, EnsemBlue makes CEDs self-describing. Each CED locally stores metadata files that contain all the information needed for an EnsemBlue client to attach and interact with that CED. For each file system object on the CED that is replicated in the Ensem-Blue namespace, EnsemBlue stores a receipt on the CED that describes how the state of the object on the CED relates to the state of a corresponding object in the EnsemBlue namespace. EnsemBlue also stores device-level metadata on the CED that uniquely identify the CED and describe its policies for propagating updates between its local namespace and the EnsemBlue namespace. Since these metadata files are small, Wolverine improves performance by reading them and caching them in memory when a CED attaches.

5.3 Supporting namespace diversity

EnsemBlue supports *namespace diversity*. It maintains a common distributed namespace that the user can organize — this namespace is exported by all general-purpose clients. On a CED that mandates a custom organization, EnsemBlue stores data in a local namespace that matches the mandated organization.

EnsemBlue views files stored on a CED as replicas of files in its distributed namespace. Each receipt maintains a one-to-one correspondence between the file in the CED namespace and the file in the distributed namespace. It stores the fully-qualified pathname of the file in the local namespace and its unique EnsemBlue identifier.

A receipt can be viewed as a type of symbolic link since it relates two logically equivalent files. We considered using symbolic links directly. However, if such links were to reside in EnsemBlue, a disconnected client would be unable to interpret the files on a CED if it did not have all relevant links cached when the CED attached. The alternative of using symbolic links in the CED's local file system is unattractive because many CED file systems do not support links. Receipts avoid both pitfalls since they are file system independent and reside on the CED.

Each receipt also contains version information. For the local namespace, the receipt stores a modification time. For the EnsemBlue namespace, the receipt stores the version vector described in Section 6. When a CED attaches to a general-purpose client, Wolverine detects modifications by comparing the versions in the receipt with the current version of the file in both namespaces. The next two subsections describe how it propagates updates between namespaces when versions differ.

5.4 Reintegrating changes from CEDs

On a general-purpose EnsemBlue client, a kernel module intercepts file modifications and redirects them to Wolverine. However, most CEDs do not allow the insertion of kernel modules, making this method infeasible. Thus, for a closed-platform CED, Wolverine uses a strategy similar to the one used by file synchronization tools such as rsync [26] and Unison [17] in which it scans the local file system of the CED to detect modifications. Wolverine scans a CED when it is first attached and subsequently at an optional device-specific interval.

When a CED attaches to a general-purpose client, Wolverine lists all files on the CED using the interface exported by that device. For instance, for CEDs that export a file system interface, Wolverine does a depth-first scan from the file system root. Usually, this scan is quick: on an iPod mini with 540 MP3s comprising 3.4 GB of storage, the scan takes less than 2 seconds. If a file on the CED has a modification time later than the time stored in its receipt, the file has been modified since the last time the CED detached from an EnsemBlue client. Wolverine copies the file from the CED to the EnsemBlue namespace and updates its receipt.

If a file or directory is found on the CED for which no receipt exists, Wolverine creates a corresponding object in the EnsemBlue namespace. It first retrieves the receipt of the object's parent directory on the CED. From the receipt, it determines the EnsemBlue directory that corresponds to the parent directory. It replicates the new object in that EnsemBlue directory.

To bootstrap this process, a user associates the CED with an EnsemBlue directory the first time the CED is attached. Wolverine replicates the local file system of the CED as a subtree rooted at the specified directory. The user can then reorganize the data by moving files and directories from that subtree to other parts of the EnsemBlue namespace. Moving objects within EnsemBlue does not affect the organization of files on the CED.

For example, a cell phone user might download MP3s from a content provider and take pictures with a built-in camera. If the phone is associated with the directory /ensemblue/phone and the phone has separate music and photos subdirectories, EnsemBlue creates two directories /ensemblue/phone/photos and /ensemblue/phone/music that contain the new content. The user may change this behavior by moving the directories within EnsemBlue; e.g., to subtrees that store other types of music and photos. Not only will the files currently in these directories be moved to the new location, but future content created by the cell phone will be placed into the directories at their new locations.

The user can exert fine-grained control over the placement of data in EnsemBlue by relocating individual files. For instance, the user might move MP3s into a directory structure organized by artist and album. Since moving each file manually would be quite tedious, one can use persistent queries to automate this process. For instance, a music organizer could create a query to learn about new files that appear in the /ensemblue/phone/music directory. For each file, it would read the artist and album from the ID3 tag, then move the file to the appropriate directory (creating the directory if needed). In this example, the combined automation of namespace diversity and persistent queries lets the user exert substantial control over the organization of data with little effort.

If a file is deleted from a CED, Wolverine detects that a receipt exists without a corresponding local file. Depending on the policy specified for the device, Wolverine may either delete both the receipt and its corresponding file in the EnsemBlue namespace, or it may delete only the receipt. We have found the latter policy appropriate for CEDs such as DVRs and cameras on which files are often deleted due to storage constraints.

5.5 Propagating updates to CEDs

Modifications made to files in EnsemBlue are automatically propagated to corresponding files in local CED namespaces. When Wolverine creates a receipt for an object stored on a CED, it also sets a callback with the server for that file on behalf of the CED. If the file is subsequently modified by another client, the server sends an invalidation to the client to which the CED is currently attached (this client may be different from the one that set the callback). Upon receiving a callback, Wolverine fetches the new version of the file and updates the replica

and its receipt on the CED. If the CED is not attached to a client when a file is modified, the server queues the invalidation and delivers it when the CED next attaches.

EnsemBlue uses affinity to determine whether the local namespace of a CED should be updated when a new file is created. The user may specify that a subtree of the EnsemBlue namespace has affinity with a directory in the local CED namespace. At that time, Wolverine replicates the subtree in the specified directory on the CED. When setting affinity, the user may optionally specify that files created in the future in the EnsemBlue subtree should also be replicated on the CED.

CEDs support type-specific affinity. A command line tool lets the user create a persistent query as a hidden file in any directory on the CED. The EnsemBlue server initially appends event records for all existing files that match the specified query. When a file matching the query is created, the server appends an additional record. Since a callback is set on behalf of the CED for the new query, the client to which the CED is attached receives an invalidation when the server inserts new records in the query. Wolverine fetches the file referenced by each record and creates a corresponding file in the CED directory. In this manner, the CED directory is populated with all files that match the query. This use of typespecific affinity is inspired by the Semantic File System [5]. However, in contrast to SFS where directories populated by semantic queries are virtual, EnsemBlue replicates data on the CED so that the results of a query are available when the CED is disconnected.

6 Ensemble support

EnsemBlue supports ensembles, which are collections of devices that share a common view of the file system over a local network. Ensembles allow clients disconnected from the file server to share their cache contents and propagate updates to each other. For instance, a mobile user who lacks Internet access may carry a laptop, a cell phone, and an MP3 player. EnsemBlue lets these devices share a mutually consistent view of the distributed namespace. Data modifications made on one client will be seen on the others. Only clients that share a common server can form an ensemble (such devices would typically be owned by the same user or family). A client joins only one ensemble at a time.

6.1 Design considerations

In designing support for ensembles, we wished to reach a middle ground between file systems such as BlueFS [14] and Coda [10] that support disconnected operation but do not let two disconnected clients communicate, and systems such as Bayou [25] that eliminate the file server

and propagate data only through peer-to-peer exchanges. There is an important role for a central repository of data in personal file systems. A system that stores personal multimedia is entrusted with data such as family photos that have immense personal significance. Storing the primary replica of such files at the server ensures that one copy is always in a reliable location. The server is also a highly-available location from which any client may obtain a copy of the file. Yet, the peer-to-peer model is appealing in the ensemble environment. As the number of personal computing devices increases, a mobile user will often have two or more co-located devices that are disconnected from the server. Devices that cache content of interest to others should be able to share their data.

One important difference between ensembles and peerto-peer propagation is the length of interaction. Bayou devices communicate through occasional synchronization: two clients come into contact, exchange updates, and depart. Ensembles, in contrast, allow long-lived interactions among disconnected devices. A client that joins an ensemble first reconciles the state of its cache with the view of the file system shared by the ensemble. It participates in the ensemble until it either loses contact with the other devices or reconnects with the server.

One general-purpose computer, called the *castellan*, acts as a pseudo-server for the ensemble. The castellan maintains a *replica list* that tracks the cache contents of each member. When a member suffers a cache miss, it contacts the castellan, which fetches the file from another member, if possible. Clients also propagate updates to the castellan when they modify files — the castellan in turn propagates the modifications to interested clients within the ensemble. For example, a PDA might be used to display photos taken with a cell phone. The cell phone updates a shared directory when it takes a new photo, causing the castellan to invalidate the replica of the directory cached on the PDA. Subsequently, the PDA would contact the castellan to fetch the modified directory and new photo from the phone.

Ensembles are designed to support specialized CEDs that cache only a small portion of the objects in the file system. Devices such as MP3 players and cell phones are incapable of storing a complete copy of all objects in a data volume, as is done in systems such as Bayou, or keeping a log of updates to all objects, as is done in systems such as Footloose [15] and Segank [22]. For instance, a single video exceeds the storage capacity of a typical cell phone. EnsemBlue lets a client cache only the objects with which it has affinity. Thus, a CED need not waste storage, CPU cycles, or battery energy processing updates for files it will never access.

We struggled to balance the concerns of consistency and availability. When a client is disconnected from the

server, its cached version of a file may be stale since it cannot receive invalidations. Other clients within the ensemble would see the stale version if they read the file, assuming that no other member cached a more recent version. In the worst case, a client may have previously seen an up-to-date version of the file, evicted that version from its cache, then joined the ensemble while disconnected. The client would see an older version of the file than it previously viewed under this scenario.

We initially devised many solutions to improve consistency in ensembles. For example, we considered having each client remember the versions of all objects that it ever read. We also considered having each client store complete version information for all objects in the file system. However, we felt that these solutions did not fit well with a storage system that focuses on personal multimedia. The types of file updates that we envision a user making while disconnected are often trivial; e.g., changing the rating on a song, or adding a tag to a photo. We therefore asked ourselves: is it better to present data that might possibly be stale or to present no data at all? In our target environment, we believe the former answer is correct (although in a workstation environment, we would give a different answer).

6.2 Implementation

An ensemble is formed when two or more disconnected clients share a local area network. For wireless devices, we leverage PAN-on-Demand [1], which lets devices owned by the same user discover each other and form a personal area network. Since PAN-on-Demand targets devices carried by a single user, it assumes that its members can communicate via a single radio hop. To form an ensemble, at least one client must be a general-purpose computer. This device serves as the castellan; the other devices are its clients. CEDs can join an ensemble by attaching to a general-purpose ensemble member.

6.2.1 Tracking modifications

EnsemBlue tracks modifications for each object using a version vector [16] that contains a <client, version> tuple for each client that modified the object. A modifying client increments the version in its tuple. If it has not yet modified the object, it appends a tuple with its unique EnsemBlue client identifier and a version of one.

Version vectors are used to compare the recency of replicas. If two version vectors are the same, the replicas are equivalent. For non-equivalent replicas, we say that a replica is more recent than another if for every client in the version vector of the second replica, the client exists in the version vector of the first replica with an equal or greater version number. If two replicas are not equivalent and neither is more recent than the other, concurrent updates have been made by different clients. In this case, EnsemBlue asks the user to manually resolve the conflict. However, if concurrent updates have been made to a directory and EnsemBlue can automatically merge the updates, it will do so without user involvement; e.g., it will merge updates that create two different files in the same directory. This strategy is the same as Coda's strategy for reintegrating changes from disconnected clients.

While a client is connected to the server, its locally cached replicas are the same as the primary replicas stored on the server. Once a client disconnects, it appends operations that modify its cached objects to a *disconnection log*. When the client reconnects with the server, it replays the disconnection log to reintegrate changes. Each disconnection log entry contains a modification (write, file creation, rmdir, etc.), the version vectors of all objects modified, and the unique identifier of the client that made the modification.

Each client maintains the invariant that for each cached object, the disconnection log contains all operations needed to recreate the cached version of the object starting with a version already seen by the server. A client that never joins an ensemble maintains this invariant trivially since it appends an entry to the disconnection log every time it modifies an object. When it disconnected, all cached objects were versions seen by the server. By replaying the log, the server can reconcile each primary replica with the modified version on the client.

6.2.2 Joining an ensemble

A disconnected client joins an ensemble when it discovers another disconnected client or an existing ensemble on its local network. The joining computer becomes a client of the castellan of the existing ensemble. If the discovered device is an isolated disconnected client, then the discovered device becomes the castellan and the discoverer becomes its client. While the current method of choosing the castellan is arbitrary, selecting a less-capable device as the castellan, e.g., a PDA instead of a laptop, impacts only performance, not the correctness of the system. In the future, we plan to add heuristics that select the most capable client to be the castellan.

Before a disconnected client joins an ensemble, it must reconcile its view of the EnsemBlue namespace with that of the ensemble. After reconciliation, if an object is replicated on more than one ensemble client, all replicas are the same, most up-to-date version.

The joining client sends the castellan the version vectors of its cached objects. The castellan stores the version vectors of all objects cached on any ensemble member in the replica list. By comparing the two sets of version vectors, it first determines the set of objects that are replicated on both the ensemble and the joining client. It then

determines the subset of these objects for which the ensemble version and the version on the joining client differ — this is the set of objects to be reconciled.

If an object being reconciled is in conflict due to concurrent updates on disconnected clients, the user must resolve the conflict before the new device joins the ensemble. Otherwise, EnsemBlue brings the less recent replica up-to-date. If the disconnection log on the client with the more recent replica contains sufficient records to update the less recent replica, the log records are transmitted to the out-of-date client. The client applies the logged operations and adds the records to its disconnection log. If the ensemble version is out-of-date, the castellan applies the log records to its local cache if necessary, then forwards the records to other members that cache the object. We anticipate that transmitting and applying log records will usually be more efficient than transmitting the entire object. Most multimedia files are large, yet updates are often trivial; e.g., setting the fields in an ID3 tag.

Sometimes, log records alone are insufficient to bring the less recent replica up-to-date. This occurs when the less recent replica is older than the version associated with any record in the disconnection log of the client with the most recent replica. In this case, EnsemBlue simply invalidates the stale replica. Any client that has affinity to the invalidated object fetches the most recent version and its associated log records after the ensemble forms.

Reconciliation ends when all out-of-date replicas on the joining client and the existing ensemble clients have been updated or invalidated. The castellan updates the replica list to include objects cached by the joining client.

6.2.3 Ensemble operation

After joining an ensemble, a client can fetch data from other ensemble members via the castellan. EnsemBlue inherits the dynamic cache hierarchy of BlueFS [14]. It fetches data from the location predicted to have the best performance and use the least energy. A client that decides to fetch data from the ensemble sends an RPC to the castellan. The castellan examines the replica list to determine which client, if any, caches the data. It either services the request itself or forwards it to another client. If the data is not cached within the ensemble, the castellan returns an error code.

If the requesting client does not have an up-to-date version of the object, the responding client includes all records in its disconnection log that pertain to the object—this maintains the invariant described in Section 6.2.1. The requesting client appends the records to its disconnection log and caches the object in its local storage. The castellan updates its replica list to note that the object is now cached by the requesting client.

Any ensemble client that modifies an object sends an RPC to the castellan. The castellan forwards the modification to all ensemble members that cache the object. These clients update their cached objects and append a record to their disconnection logs. Thus, most clients are only informed of updates that are relevant to them. The castellan does not send clients updates and requests for objects they do not cache. For instance, an MP3 player that caches only music files will not be informed about updates to photos or asked to service requests for e-mail.

6.2.4 Leaving an ensemble

When the castellan loses contact with a client, the client is considered to have left the ensemble. The castellan removes the objects cached by that client from the replica list. If the castellan departs, the ensemble is disbanded. The remaining clients form a new ensemble if they remain in communication.

A client that leaves an ensemble operates disconnected until it joins another ensemble or reconnects with the server. Its disconnection log may now contain updates made by other clients. Upon reconnection, the client sends its log to the server. Since the first entry in the log for any object modifies a version previously seen by the server, the server can use the log to update the primary replicas. However, since multiple clients can reintegrate the same log record, the server must not apply the same record twice. It uses the version vectors in the log records to eliminate duplicates. If the version that it caches is more recent than the log record, it ignores the modification. Otherwise, it applies the modification to the primary replicas. If a primary replica is more recent than the replica on the reconnecting client, the server sends the client an invalidation.

This design lets mobile clients reconcile changes on behalf of other clients. Thus, a client can remain up-to-date even though it never reconnects with the server. For example, a car stereo could retrieve MP3s from a disconnected client such as a laptop that is transported in the car. The stereo could also reintegrate changes to play lists and song ratings back to the server via the laptop.

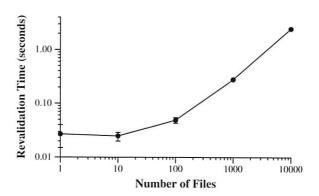
7 Evaluation

Our evaluation answers the following questions:

- What is the overhead of forming an ensemble?
- What is the overhead of persistent queries?
- How effectively can CEDs such as cameras and MP3 players be integrated with EnsemBlue?

7.1 Ensemble formation

We measured the time two disconnected clients take to form an ensemble. During this experiment, an IBM X40



This figure shows how the time to form an ensemble varies with the number of files stored on each device. The graph is log-log. Each result is the mean of 7 trials — the error bars are 90% confidence intervals

Figure 2. Ensemble formation time

laptop with a 1.2 GHz Pentium M processor and 768 MB of RAM becomes the castellan, and an IBM T20 laptop with a 700 MHz Pentium 3 processor and 128 MB of RAM becomes its client. The computers communicate via an 802.11b wireless ad-hoc connection.

Figure 2 shows how the time to form the ensemble changes as we vary the number of files cached on both clients. This data is displayed using a log-log graph due to the disparity in reconciliation time. At the beginning of each experiment, both laptops cache the same version of each file. Since only metadata is exchanged in this experiment, file size is unimportant and all files are zero length. In the absence of out-of-date replicas, the time to form an ensemble is quite small. Beyond an approximately 20 ms constant performance cost for the initial message exchange, formation time is roughly proportional to the number of cached items. Even with 10,000 files cached on both machines, the ensemble forms in less than three seconds.

We next measured the effect of reconciling out-of-date objects on ensemble formation time. At the beginning of this experiment, the X40 client contains 18 MP3 files that total 115 MB in size, as well as 40 photos comprising a total of 110 MB of data. The T20 contains only the 18 MP3 files. It does not have affinity for the photos and does not cache them.

We consider the four scenarios in Figure 3. In the first scenario, none of the files are modified. As predicted by the previous experiment, the ensemble is formed in a fraction of a second. In the second scenario, the X40 modifies the ID3 tag of all MP3s prior to joining the ensemble. The ensemble is formed in slightly less than a second. The additional delay reflects the time for the X40 to transmit its disconnection log records that correspond to the ID3 tag modifications.

Scenario	Formation time (seconds)	
None	0.34 (0.32–0.36)	
ID3 Tags	0.95 (0.91-0.96)	
Music	226 (219–233)	
All	227 (221-234)	

This figure shows how the number of updates reconciled affects ensemble formation time. In the first row, no files are updated. In the second row, ID3 tags on 18 MP3s are updated. In the third row, the 18 MP3s are re-encoded, and in the last row, the 18 MP3s and 40 photos are completely modified. Each result is the mean of 5 trials — minimum and maximum values are in parentheses.

Figure 3. Time to reconcile updates in an ensemble

In the third scenario, the X40 overwrites the contents of all MP3s before joining the ensemble — this corresponds to a user re-encoding the MP3s from a CD. It takes 227 seconds to form the ensemble since each large file on the T20 is completely overwritten. For comparison, the time to manually copy the files is 209 seconds. Thus, the overhead due to EnsemBlue is only 8%. The difference between the second and third scenarios illustrates the benefit of shipping log records rather than entire objects during reconciliation. When modifications are a small portion of the total size of each file, EnsemBlue realizes a substantial performance improvement over a manual copy of the files.

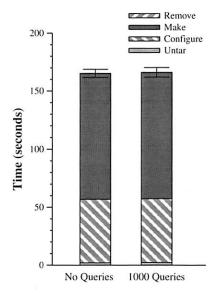
In the final scenario, the X40 overwrites both the MP3 files and the photos in its cache. However, the time to form the ensemble is virtually identical to the third scenario. Since the T20 does not cache photos, it does not have to be informed about the additional updates; the X40 ships only log records that pertain to the MP3 files. From these results, we conclude that EnsemBlue can achieve substantial performance benefit by limiting reconciliation to the set of objects cached on both clients. In contrast, a file system that transfers all updates would nearly double the reconciliation time.

7.2 Persistent query overhead

We next measured the overhead of persistent queries, which is exhibited in two ways. First, evaluating a large number of queries might slow the server as it processes modifications. Second, the evaluation of individual queries might exhibit a high latency that would preclude them from being used by interactive applications.

For these experiments, the EnsemBlue server was a Dell Precision 370 desktop with a 3 GHz Pentium 4 processor and 2 GB of RAM. The client was the X40 laptop from the previous experiments. The two computers are connected via 100 Mb/s Ethernet.

To evaluate the first source of overhead, we ran an I/O intensive benchmark in which we untar the Apache 2.0.48



This figure compares the time to run the Apache build benchmark with no queries outstanding and 1,000 queries outstanding. Each result is the mean of 5 trials — the error bars are 90% confidence intervals.

Figure 4. Overhead of persistent queries

source tree into EnsemBlue, run configure in an object directory within EnsemBlue, run make in the object directory, and finally remove all files. Figure 4 compares the time to perform the benchmark when the server is evaluating 1000 outstanding queries with the time to perform the benchmark when no queries are outstanding. The results are identical within experimental error, indicating that persistent query evaluation is not a substantial source of overhead.

To evaluate the second source of overhead, we measured the time to create a query while varying the number of records returned. Before running each experiment, we populated EnsemBlue with the data from the personal volume of a user of the prototype described in Section 2. This data set contains 5,276 files and is over 7 GB in size. We created persistent queries with the initial bit set in the event mask and with the query string matching the various file types shown in Figure 5.

We measured the time for an application to create each query and read all matching records. The results show a fixed cost of approximately 126 ms. While the latency increases with the number of matching records, the incremental cost is small. If all 5,276 files match the query string, the experiment takes 62 ms longer.

From these results, we conclude that persistent queries impose minimal overhead during both creation and evaluation. We are encouraged that these results indicate that persistent queries are cheap enough to be employed by a wide variety of applications.

File type	Matches	es Creation time (seconds)		
None	0	0.126 (0.125-0.126)		
TiVo	2	0.126 (0.125-0.126)		
text	45	0.128 (0.126-0.133)		
jpeg	132	0.127 (0.126-0.128)		
postscript	712	0.135 (0.116-0.146)		
MP3	1729	0.150 (0.147-0.160)		
All	5276	0.188 (0.161-0.198)		

This figure shows the time to create a persistent query matching varied numbers of files. Each result is the mean of 8 trials — the values in parentheses are the minimum and maximum trials.

Figure 5. Time to create a persistent query

7.3 Case study: Integrating a digital camera

In the next two sections, we present case studies in which we examine how well CEDs can be integrated with EnsemBlue. In the first, we take pictures using a Canon PowerShot S40 digital camera. The camera produces JPEG photos that it stores in a FAT file system. We registered the camera with EnsemBlue by specifying a root directory to which photos should be imported. The camera groups the photos it takes into subdirectories that contain 100 photos each. The default behavior of EnsemBlue is to recreate the camera directory structure within the root directory specified during registration. This is not the most user-friendly of organizations.

We first decided to organize our photos by date. The camera stores metadata in each JPEG that specifies when it took the photo. We created a photo organizer application that creates a persistent query to be notified when a new photo is added to EnsemBlue. The organizer reads the JPEG metadata and moves the file to a subdirectory specific to that date, creating the directory if necessary. After moving each JPEG file, the organizer removes the notification from the query.

We next enhanced our organizer to arrange photos by appointment. The organizer takes as a parameter the location of an ical .calendar file. When a new JPEG is added, it searches the file for any appointment entered for the time the photo was taken. If it finds such an appointment, it moves the photo to a subdirectory for that appointment within the directory for the date when the photo was taken (again, creating the subdirectory as needed). The photo organizer required only 123 lines of code, indicating that such applications can be created with minimal effort by CED manufacturers, third-party software developers, and technically-savvy users.

We measured the time to import photos from our camera into EnsemBlue using the same experimental setup as in the previous section. The camera, containing 201 new photos approximately 256 MB in size, is attached to the

X40 laptop client. EnsemBlue takes approximately 188 seconds to import the photos. In contrast, copying all the files manually takes 174 seconds. The approximately 7% overhead imposed by EnsemBlue seems a reasonable price to pay for automatic replication and organization of the imported photos, especially when one considers the time required to manually organize 201 images.

7.4 Case study: Integrating an MP3 player

Our second case study integrates an iPod mini MP3 player and a D-Link media player with EnsemBlue. The iPod is a mobile device that stores and plays music files of several different formats. It presents two challenges for integration with a distributed file system. First, music files are stored in specific subdirectories of its local file system — the music files should be spread between these subdirectories to improve lookup latency. Second, the iPod uses a custom database to store information about the music files in its local storage. This database must be updated when files are added.

We address the first challenge with type-specific affinity. To place files in specific subdirectories on the iPod, we create a query for each directory that matches music files stored within EnsemBlue. To divide files between subdirectories, we take the simple approach of partitioning the namespace by filename. For example, the query for the first subdirectory matches on music files that begin with 'a'. When a new music file beginning with 'a' is added to EnsemBlue, the server inserts a record in the query for that directory. When the iPod next attaches to a client, that client fetches the query, reads the record, and replicates the file on the iPod within that subdirectory.

We address the second challenge by creating a standalone application to update the iPod database. This application creates a query that matches all music files. When a file is added, the application updates the iPod database within the EnsemBlue namespace using Gnu-Pod. The database on the iPod's local storage has affinity to this file. Thus, when the iPod attaches to a client, that client receives a callback on behalf of the iPod for the database file. It fetches the database and replicates it on the iPod. This application required only 86 lines of code.

We also added support for a D-Link media player that can only play music files encoded in the MP3 format. Since many of our music files are encoded in the M4A format, we wrote a transcoder, described in Section 4.3, to convert files so that they could be played on the media player. The D-Link media player can read files using a client program that exports the file system of a general-purpose computer. Thus, we simply run that program on an EnsemBlue client; the media player can play music in EnsemBlue without further customization. One of our

group members uses an identical strategy to play music files stored in EnsemBlue on a TiVo DVR.

8 Related work

To the best of our knowledge, EnsemBlue is the first distributed file system to provide explicit support for consumer electronics devices. Its novel contributions include persistent queries, which leverage the underlying cache consistency mechanisms of the file system to deliver application-specific event notifications, and receipts, which allow namespace diversity.

Many operating systems provide application notifications about changes to local file systems; Linux's inotify [12], the Windows Change Journal [4], and Apple's Spotlight [24] are three examples. Watchdogs [2], proposed by Bershad and Pinkerton, combine notification with customization by allowing user-level applications to safely overload file system operations for particular files and directories. Unlike persistent queries, these mechanisms are limited to a single computer and do not scale to the distributed environment targeted by EnsemBlue. Our approach of evaluating persistent queries on the server realizes the same benefit of pushing evaluation to the data that has been previously shown in projects such as Active Disks [19] and Diamond [8]. Salmon et al. [20] have recently proposed views that allow pervasive devices to publish interest in particular sets of objects with peer devices. Views are similar to persistent queries in that they allow clients to specify the objects in which they are interested as a semantic query. Since views target a serverless system, each view must be propagated to all peers.

Type-specific affinity is similar to the virtual directories presented by the Semantic File System [5]. However, the directory contents produced by type-specific affinity are persistent, meaning that they can be accessed by a mobile computer or CED that is disconnected from the file server. Persistent queries should not be confused with applications such as Glimpse [13] and Connections [23] that index file system data. Rather, persistent queries are a tool that such applications can use; they notify indexing applications about changes to file system state.

EnsemBlue's strategy of scanning the local file system of closed-platform CEDs is similar to the way that file synchronizers operate [17, 26]. The main difference between the two strategies lies in EnsemBlue's use of receipts. Receipts are a more general mapping between namespaces that allow files to be moved within the distributed namespace, yet retain the same location in the local device namespace. As shown in Section 7.3, receipts can be combined with persistent queries to automate the remapping of individual files between namespaces.

EnsemBlue's model of supporting isolated disconnected clients owes much to Coda [10]. From Coda, EnsemBlue inherits the use of a disconnection log to store updates, as well as its conflict resolution strategy. However, EnsemBlue differs from prior distributed file systems in its explicit support for ensembles of disconnected clients. While others have identified ensembles as an emerging paradigm for personal mobile computing [21], EnsemBlue is the first server-based distributed file system to explicitly support this model of computing.

Many previous storage systems have eschewed a central server in favor of propagating data through peer-to-peer exchanges. Bayou [25] replicates data collections in their entirety. A Bayou client can read and write any accessible replica. Replicas are reconciled by applying perwrite conflict resolution based on client-provided merge procedures. EnsemBlue differs from Bayou in that it allows its clients to cache only a portion of a data volume. Further, ensembles remain consistent after formation, whereas Bayou replicas can diverge after each reconciliation.

Footloose [15] and EnsemBlue both present a consistent view of objects on devices located close to the user. Like Bayou, Footloose allows clients to exchange data through propagation of update records. Wishes in Footloose provide an analogous event notification mechanism to persistent queries. However, unlike persistent queries, wishes do not guarantee at-least-once delivery. Footloose requires that update records be preserved until every interested client is known to have received the record; this could be a problem for CEDs with limited storage. In contrast, EnsemBlue clients can discard update records once they have been reconciled with its server. Footloose targets CEDs as clients, but requires that any client be sufficiently open to run their Java code base. Footloose does not export a file system interface, and thus cannot be used with legacy applications.

Other systems that support peer-to-peer update propagation are Segank [22], in which a MOAD carried by a user at all times ensures consistency among computers that share a namespace, Ficus [6], Files Every Where [18], and OmniStore [9].

9 Conclusion

Consumer electronics devices are increasingly important computing platforms. Yet, it remains challenging to integrate them into existing distributed systems. CED architectures are typically closed, admitting only a narrow interface for interaction with the outside world. Their capabilities are non-uniform since available resources are chosen to support a particular application. This heterogeneity implies that a distributed system that supports

CEDs should be flexible. It must customize its interactions with each device according to the interface presented. If a CED lacks the necessary resources to participate in a distributed protocol, the protocol should allow for other, more general-purpose participants to supply needed resources on its behalf.

EnsemBlue shows the benefit of flexibility. By supporting namespace diversity, device ensembles, and persistent queries, EnsemBlue is highly extensible. As the case studies in this paper demonstrate, extensibility is crucial to making devices such as MP3 players, cameras, and media players full-fledged participants in EnsemBlue. Based on these results, we are hopeful that similar principles can be applied to other distributed systems to allow them to support CEDs.

Acknowledgments

We thank Bill Schillit and Nitya Narasimhan for fruitful discussions about this topic, and Edmund B. Nightingale for his help with BlueFS. Kumar Puspesh added PTP support to EnsemBlue. Manish Anand, Ya-Yunn Su, and Kaushik Veeraraghavan, and the anonymous reviewers provided valuable feedback. The work is supported by the National Science Foundation under award CNS-0306251. Jason Flinn is supported by NSF CAREER award CNS-034686. Intel Corp and Motorola Corp have provided additional support. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, Motorola, the University of Michigan, or the U.S. government.

References

- ANAND, M., AND FLINN, J. PAN-on-Demand: Building self-organizing WPANs for better power management. Tech. Rep. CSE-TR-524-06, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, 2006.
- [2] BERSHAD, B. B., AND PINKERTON, C. B. Watchdogs extending the unix file system. *Computer Systems* 1, 2 (Spring 1988).
- [3] CALLAGHAN, B., PAWLOWSKI, B., AND STAUBACH, P. NFS Version 3 Protocol Specification. Tech. Rep. RFC 1813, IETF, June 1995.
- [4] COOPERSTEIN, J., AND RICHTER, J. Keeping an eye on your NTFS drives: the Windows 2000 Change Journal explained. *Microsft Systems Journal* (September 1999).
- [5] GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND O'TOOLE, J. W. Semantic file systems. In *Proceedings of the* 13th ACM Symposium on Operating Systems Principles (October 1991), pp. 16–25.
- [6] GUY, R. G., HEIDEMANN, J. S., MAK, W., PAGE, T. W., POPEK, G. J., AND ROTHMEIER, D. Implementation of the Ficus replicated file system. In *Proceedings of the Summer USENIX Conference* (June 1990), pp. 63–71.
- [7] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. ACM Transactions on Computer Systems 6, 1 (February 1988).
- [8] HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARYANAN, M., GANGER, G. R., RIEDEL, E., AND AILAMAKI, A. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the USENIX FAST* '04 Conference on File and Storage Technologies (March 2004).

- [9] KARYPIDIS, A., AND LALIS, S. Omnistore: A system for ubiquitous personal storage management. In *Proceedings of the* 4th IEEE International Conference on Pervasive Computing And Communications (Pisa, Italy, March 2006), pp. 136–147.
- [10] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. ACM Transactions on Computer Systems 10, 1 (February 1992).
- [11] http://lame.sourceforge.net/.
- [12] LOVE, R. Kernel korner: Intro to inotify. Linux Journal 2005, 139 (2005), 8.
- [13] MANBER, U., AND WU, S. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the 1994 Winter USENIX Conference* (San Francisco, CA, January 1994), pp. 23–32.
- [14] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the* 6th Symposium on Operating Systems Design and Implementation (San Francisco, CA, December 2004), pp. 363–378.
- [15] PALUSKA, J. M., SAFF, D., YEH, T., AND CHEN, K. Footloose: A case for physical eventual consistency and selective conflict resolution. In *Proceedings of the 5th IEEE Workshop on Mobile* Computing Systems and Applications (Monterey, CA, Oct. 2003).
- [16] PARKER, D. S., POPEK, G. J., RUDISIN, G., STOUGHTON, A., WALKER, B. J., WALTON, E., CHOW, J. M., EDWARDS, D., KISER, S., AND KLINE, C. Detection of mutual inconsistencies in distributed systems. *IEEE Transactions on Software Engineer*ing SE-9, 3 (May 1983), 240–247.
- [17] PIERCE, B. C., AND VOUILLON, J. What's in Unison? A formal specification and reference implementation of a file synchronizer. Tech. Rep. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.
- [18] PREGUICA, N., BAQUERO, C., MARTINS, J. L., SHAPIRO, M., ALMEIDA, P. S., DOMINGOS, H., FONTE, V., AND DUARTE, S. Few: File management for portable devices. In *Proceedings* of the International Workshop on Software Support for Portable Storage (San Francisco, CA, March 2005), pp. 29–35.
- [19] RIEDEL, E., FALOUTSOS, C., GIBSON, G. A., AND NAGLE, D. Active disks for large-scale data processing. *IEEE Computer* (June 2001), 68–74.
- [20] SALMON, B., SCHLOSSER, S. W., AND GANGER, G. R. Towards efficient semantic object storage for the home. Tech. Rep. CMU-PDL-06-103, Carnegie Mellon University, May 2006.
- [21] SCHILIT, B. N., AND SENGUPTA, U. Device ensembles. Computer 37, 12 (December 2004), 56–64.
- [22] SOBTI, S., GARG, N., ZHENG, F., LAI, J., SHAO, Y., ZHANG, C., ZISKIND, E., KRISHNAMURTHY, A., AND WANG, R. Y. Segank: A distributed mobile storage system. In *Proceedings of the 3rd Annual USENIX Conference on File and Storage Technologies* (San Francisco, CA, March/April 2004).
- [23] SOULES, C. A. N., AND GANGER, G. R. Connections: using context to enhance file search. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 119–132.
- [24] Spotlight overview. Tech. Rep. 2006-04-04, Apple Corp., Cupertino, CA, 2006.
- [25] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operat*ing Systems Principles (Copper Mountain, CO, 1995), pp. 172– 182.
- [26] TRIDGELL, A., AND MACKERRAS, P. The rsync algorithm. Tech. Rep. TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra, Australia, 1996.
- [27] WEISER, M. The computer for the 21st century. ACM SIGMO-BILE Mobile Computing and Communications Review 3, 3 (July 1999), 3–11.

Persistent Personal Names for Globally Connected Mobile Devices

Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, Robert Morris

Massachusetts Institute of Technology

Abstract

The Unmanaged Internet Architecture (UIA) provides zero-configuration connectivity among mobile devices through personal names. Users assign personal names through an ad hoc device introduction process requiring no central allocation. Once assigned, names bind securely to the global identities of their target devices independent of network location. Each user manages one namespace, shared among all the user's devices and always available on each device. Users can also name other users to share resources with trusted acquaintances. Devices with naming relationships automatically arrange connectivity when possible, both in ad hoc networks and using global infrastructure when available. A UIA prototype demonstrates these capabilities using optimistic replication for name resolution and group management and a routing algorithm exploiting the user's social network for connectivity.

1 Introduction

Network-enabled mobile devices such as laptops, smart phones, media players, personal digital assistants, gaming consoles, and digital cameras are becoming ubiquitous in the lives of ordinary people. The proliferation of these devices makes secure global peer-to-peer connectivity between them increasingly important. While on a trip, for example, a user in a cyber cafe may wish to copy photos from his WiFi-enabled camera to his PC at home for storage and backup. Two users who meet in a park or other off-Internet location may wish to connect their WiFi devices to exchange photos or other information, and later re-establish a connection between the same devices over the Internet after returning to their homes, without the risk of a third party intercepting the connection. A Voice-over-IP user would like his WiMax phone to be easily reachable by his friends wherever he and they are located, but not to be reachable by telemarketers.

Convenient global communication over the Internet, however, currently requires the target device to have both a global name and a static, public IP address. Users must register with central naming authorities to obtain global names, and mobile personal devices usually have dynamic IP addresses behind firewalls or network address translators [27]. Protocols such as Dy-

namic DNS [49], Mobile IP [37], and Virtual Private Networks [22] provide piecemeal solutions to these problems, but the configuration effort and technical expertise they require makes them deployable only by organizations with dedicated network administration staff. User interface refinements alone cannot overcome this deployment roadblock, because the protocols depend on centralized resources—global domain names and static, public "home" IP addresses—that are not part of most consumer-oriented Internet service packages. Ordinary users require a solution that "just works."

The *Unmanaged Internet Architecture* (UIA) is a peer-to-peer connectivity architecture that gives nontechnical users a simple and intuitive way to connect their mobile personal devices via convenient *personal names* organized into *personal groups*. A user can *merge* multiple UIA devices to form a personal group, after which the devices work together to offer secure remote access to any device in the group from any other. The devices forming the group present the user with a shared personal namespace, which they optimistically replicate [26, 28, 47] to ensure constant availability on each device whether on or off the Internet. The devices gossip namespace changes as connectivity permits [12], and can propagate updates via mobile devices carried by the user [36].

UIA interprets personal names relative to personal groups, so users can assign concise, meaningful names like ipod instead of long globally unique names like ipod.alicesm5186.myisp.com. In this way UIA conforms to the intuitive model with which users already manage their cell phones' address books. Users normally create personal names by *introducing* devices locally, on a common WiFi network for example. Once created, these names remain persistently bound to their targets as devices move. Personal names are intended to supplement and not replace global DNS names [33]: users can refer to personal names like phone alongside global names like usenix.org in the same applications.

Different users can introduce their devices to name other users and link their respective personal groups. Bob can refer to his friend Alice as Alice, and if Alice calls her VoIP phone phone then Bob can make calls to Alice's phone using the name phone. Alice. In this way, UIA adapts peer-to-peer social networking ideas previously explored for other purposes [10, 29, 31, 38, 39]

to form a user-friendly peer-to-peer naming infrastructure. Users can also create and collect names into ad hoc *shared groups* to reflect common interests or informal organizations.

UIA devices cooperate in an overlay routing protocol to provide robust location-independent connectivity in the face of changing IP addresses, Internet routing failures, network address translators, or isolation from central network infrastructure. Although scalable routing with location-independent node identities is inherently challenging in general [21], UIA focuses on routing among friends and nearby neighbors in the user's social network. We expect the UIA routing algorithm to scale well because each node only consumes storage and bandwidth to track other nodes in its immediate neighborhood.

UIA makes the following primary contributions, expanding on previously proposed ideas [19]. First, UIA introduces a simple and intuitive model for connecting mobile devices into *personal groups*, providing ad hoc user identities, personal names, and secure remote access, without requiring the user to manage keys or certificates explicitly. Second, UIA presents a novel gossip and replication protocol to manage the naming and group state required by this user model, adapting optimistic replication principles previously developed for file systems and databases. Third, UIA leverages social networking to create a scalable overlay routing algorithm that can provide robust connectivity among social friends and neighbors without relying on central infrastructure.

The next section introduces the operation of UIA devices from a non-technical user's viewpoint. Section 3 describes UIA's design at a high level, Section 4 presents UIA's naming system in depth, followed by the routing layer design in Section 5. Section 6 summarizes implementation status and Section 7 evaluates the performance of the prototype. Section 8 discusses future work, Section 9 presents related work, and Section 10 concludes.

2 User Experience

This section describes UIA's operating principles from the perspective of a non-technical user; later sections detail how the system provides this user experience.

2.1 Introducing Devices

A UIA device ideally ships from its manufacturer preconfigured with a name for itself such as laptop or phone, which the user can keep or change as desired. The device learns additional names as its user *introduces* it to other devices owned by the same user or different users. The introduction process assigns persistent names by which the device can securely refer to other devices.

In a typical introduction, the owner(s) of two devices bring the devices together physically and connect them to a common local-area network. Each user then invokes a local-area rendezvous tool similar to Bonjour's [2] on his device, finds the other device on the network, and selects "Introduce." Each device displays an introduction key consisting of three words chosen randomly from a dictionary, as shown in Figure 1. Each user then picks the other device's introduction key from a list of three random keys. If one of the devices has unintentionally connected to the wrong endpoint, such as an impersonator on the same network, then the matching key is unlikely to appear on the list, so the user picks "None of the above" and the introduction procedure aborts. Unlike other analogous procedures [13], UIA uses short, userfriendly "one-time" keys that only need to withstand online and not offline attacks, and its multiple-choice design prevents users from just clicking "OK" without actually comparing the keys.

Users can follow the same procedure to introduce UIA devices remotely across the Internet, as long as one device has a global DNS name or IP address and the users have a trustworthy channel through which to exchange introduction keys: e.g., a phone conversation or an authenticated chat session. We also envision alternative introduction mechanisms adapted to specific rendezvous channels such as E-mail, web sites, SMS messages, or short-range wireless links; the details of particular introduction mechanisms are not crucial to the UIA architecture.

A user can introduce UIA devices either to *merge* his own devices into a *personal group* sharing a common namespace, or to create named *links* from his own group to other users' personal groups. The following sections describe these two forms of introduction, and other important group management actions, with the help of an example scenario illustrated in Figure 2.

2.2 Device Names and Personal Groups

At Time 1 in the scenario, Bob purchases a new laptop and Internet phone, which come pre-configured with the default names laptop and phone, respectively. At Time 2, Bob uses UIA's local rendezvous tool on each device to find the other device on his home WiFi network and selects "Introduce devices" on each. Bob chooses the "Merge devices" option in the introduction dialogs (see Figure 1) to merge the devices into a personal group.

The devices in Bob's group gossip both existing names and subsequent changes to the group's namespace as physical network connectivity permits. Each device attempts to preserve connectivity to other named devices as they leave the network and reappear at other locations, without user intervention whenever possible. Bob now sees his two personal names phone and laptop on both devices, and can use these names for local and remote access. Working on his laptop at home, he uses his personal name phone to reach the phone via his home WiFi LAN. When Bob subsequently takes his laptop on a trip, he can

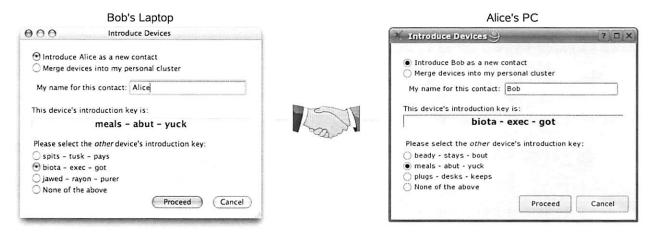


Figure 1: Bob and Alice introduce their devices

remotely access his home phone from his laptop over the Internet (e.g., to check his voice messages), still using the name phone. UIA uses cryptography to guarantee that an adversary cannot impersonate the device Bob calls phone, and cannot eavesdrop on his communication.

2.3 User Names and Social Networking

With the second form of introduction, users link their personal groups together and assign *user names* to each other, but retain exclusive control over their respective personal groups. In the example scenario, Bob purchases a new WiFi-enabled cell phone at Time 3 and meets Alice at a cafe before he has merged his cell phone with his other devices. Bob finds Alice's iPod using his cell phone's local rendezvous tool and selects "Introduce as a new contact" (see Figure 1), and Alice does likewise. Bob's phone suggests Alice's self-chosen user name Alice, but Bob can override this default (e.g., to Alice-Smith or Alice-from-OSDI) if he already knows another Alice.

Bob and Alice can now refer to each others' devices by combining device names with user names in DNS-like dotted notation. If Alice runs a web server on her home PC, named PC in Alice's personal namespace, then Bob can connect to Alice's server by typing PC.Alice into his laptop's web browser, exactly as he would use a global DNS name like usenix.org.

If Alice's personal web server is UIA-aware, she can use her name Bob in the server's access control lists so that only Bob's personal devices may browse certain private areas. UIA authenticates clients so that no one can impersonate Bob's devices to gain access to these areas.

2.4 Transitive Merging and Gossip

Bob now returns home and merges his cell phone with his home phone, as shown at Time 4 in Figure 2. Bob's home phone in turn gossips the cell phone's group membership to Bob's laptop, so the laptop and cell phone can name each other without him having to merge them explicitly. Alice's devices similarly gossip her new link named Bob and learn about Bob's three devices, after which she can, for example, refer to Bob's laptop as laptop. Bob.

Users can access or edit their personal groups from any of their devices while other devices are unreachable. If Bob and Alice are on a bus together and disconnected from the Internet, Alice can still reach Bob's laptop from her iPod via her name laptop. Bob, even if they have left their other devices at home. Bob and Alice can continue adding names for contacts they meet on the bus, and their other devices learn the new names via gossip later when they re-connect.

2.5 Resolving Conflicts

Unfortunately, both of Bob's phones happened to have identical default names of phone, resulting in their names conflicting in his newly merged namespace. UIA notifies Bob of the conflict, and he can continue using the non-conflicting name laptop, but must resolve the conflict before the name phone will work again. Bob resolves the conflict on his cell phone at Time 5, by renaming it cell while leaving the home phone with the name phone. Bob's other devices learn the resolved name bindings via gossip, as do Alice's devices, so Alice now sees Bob's phones as phone. Bob and cell. Bob.

If Bob makes conflicting namespace changes on two of his devices while they are partitioned from each other, UIA detects the conflict once the devices reconnect. Bob can continue using other non-conflicting names in the same group while conflicts exist, and he can resolve such conflicts at leisure on any of his devices.

2.6 Shared Groups

In addition to personal groups, users can create *shared groups* to help organize and share their personal names. Bob and Alice discover at Time 6 that they share an interest in photography, and decide to start a photo club for

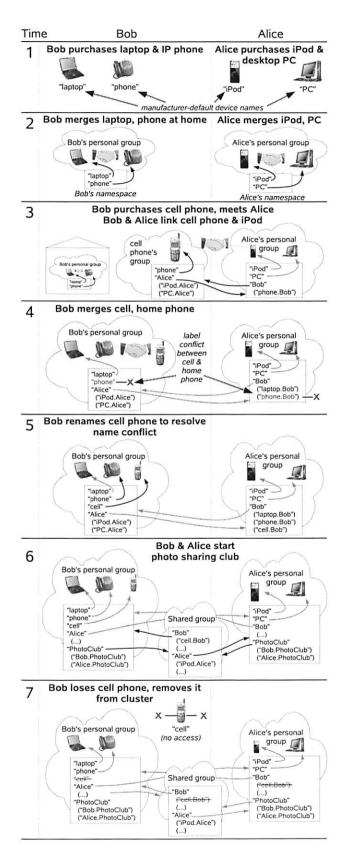


Figure 2: Example Personal Device Scenario

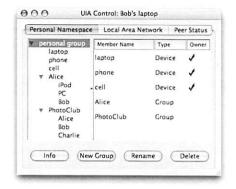


Figure 3: Groups and Ownership

themselves and other friends sharing this interest. To enable members of the club to find each other easily and share photos among their personal devices, Bob uses his laptop to create a shared group named PhotoClub in his personal namespace. On creation, the shared group's only member is Bob himself. To add Alice to the group, Bob drags the name Alice from his personal group into PhotoClub, copying his name binding for Alice into the shared group and making her the second member. Bob can similarly add other friends to PhotoClub, and these names automatically appear in Alice's view of the group the devices gossip the changes.

Although Alice can now refer to the new group as PhotoClub. Bob, she might like this group to appear directly in her own personal group instead of naming it relative to Bob. Alice drags the PhotoClub name from Bob's personal group into her own, giving herself a copy of the name leading to the same shared group. She can now refer to group members using the same names that Bob uses, such as Charlie. PhotoClub.

2.7 Group Ownership

One or more members of a UIA group may be designated as *owners*, or members allowed to modify the group. As Figure 3 illustrates, Bob's devices laptop, phone, and cell are owners of his personal group by default, allowing Bob to edit his personal group using any of his devices. The names Alice and PhotoClub are not owners, so Alice and members of PhotoClub can only browse and resolve names in Bob's namespace.

Groups can own other groups. When Bob creates his shared PhotoClub group, UIA automatically includes a name Bob in the new group that gives Bob's personal group ownership of the new group. After adding Alice to the group, Bob can give her co-ownership by clicking the owner flag by her name in the group listing, enabling her to add or remove other members herself. Ownership is transitive: Bob can modify PhotoClub using his laptop because Bob's laptop is an owner of Bob's personal group and Bob's personal group is an owner of PhotoClub.

2.8 Security and Ownership Revocation

Returning to the scenario in Figure 2, Bob loses his cell phone at Time 7, and he is not sure whether it was stolen or just temporarily misplaced. If the cell phone was stolen and has no local user authentication such as a password or fingerprint reader, the thief might obtain not only Bob's data on the cell phone itself, but also remote access to services authorized to his personal group via UIA names. UIA devices capable of accessing sensitive information remotely should therefore provide strong local user authentication, and should encrypt personal data (including UIA state) stored on the device, as Apple's File Vault does for example [3]. The details of local user authentication and encryption are orthogonal to UIA, however.

To minimize potential damage if a thief does break into Bob's user account on his cell phone, Bob can revoke the cell phone's ownership of his personal group. If the cell phone re-appears and Bob realizes that he just misplaced it, then he can "undo" the revocation and return the phone to its normal status. If the cell phone remains missing, however, UIA ensures that no one can remotely access personal information or services on Bob's other devices via the lost phone once the revocation announcement has propagated to those devices. Similarly, the cell phone loses its access to the files Alice shared with Bob as soon as Alice's PC, on which the files reside, learns of the revocation from any of Bob's remaining devices.

2.9 Ownership Disputes

Revocation cuts both ways: a thief might try to "hijack" Bob's personal group, using the stolen cell phone to revoke the ownership of Bob's other devices before Bob finds that the phone is missing. In UIA's current ownership scheme in which all owners have full and equal authority over a group, Bob's devices cannot distinguish the "real" Bob from an impostor once a stolen device's local access control is broken. UIA therefore allows any device to *dispute* another device's revocation of its ownership.

In the example scenario, when Bob next uses his laptop, UIA informs him that his laptop's ownership of his personal group has been revoked by the cell phone, which Bob realizes was stolen. In response, Bob issues a revocation of the cell phone's ownership from his laptop. The two mutual revocations effectively split Bob's original personal group into two new, independent groups: one containing only the cell phone, the other containing Bob's remaining devices. All existing UIA names referring to Bob's old personal group, and any access authorizations based on those names, become unusable and must be manually updated to point to the appropriate new group. Alice's name Bob for example is now marked "disputed" in Alice's namespace, and Alice's PC rejects attempts by any of Bob's devices to access the files she shared with Bob earlier using that UIA name. To update

her name for Bob and safely renew his access, Alice can re-introduce her devices directly to Bob's the next time they meet, or obtain a fresh link to Bob's new personal group from a trusted mutual friend who already has one.

Group ownership disputes need not be permanent. Suppose two people who co-own a shared group get into an argument, and split the group by issuing mutual revocations. If the original co-owners later settle their differences, they can undo their conflicting revocations or simply merge their respective "splinter" groups back together via UIA's normal merge mechanism. Links to the original group become unusable during the dispute, but function again normally after the dispute is resolved.

3 Basic Design

This section outlines UIA's high-level design, which consists of separate naming and routing layers that together realize the user experience described above. Sections 4 and 5 detail the naming and routing layers, respectively.

3.1 Personal Endpoint Identities

UIA devices identify each other using cryptographically unique *endpoint identifiers* or EIDs. Whereas DNS maps a name to an IP address, UIA maps a personal device name such as Bob's laptop to an EID. Unlike IP addresses, EIDs are *stable* and do not change when devices re-connect or move. UIA's routing layer tracks mobile hosts by their EIDs as they change IP addresses, and can forward traffic by EID when IP-level communication fails due to NAT or other Internet routing discontinuities.

A UIA device creates each EID it needs automatically by generating a fresh public/private key pair and then computing a cryptographic hash of the public key. As in SFS [32], EIDs are cryptographically unique, self-configuring, and self-certifying, but not human-readable. As in HIP [34], UIA-aware network transports and applications use EIDs in place of IP addresses to identify communication endpoints. (UIA can also disguise EIDs as "actual" IP addresses for compatibility with unmodified legacy applications, as described later in Section 6.)

An EID corresponds to a particular user's presence on a particular device. A user who owns or has access to several devices has a separate EID for each. A device accessed by only one user needs only one EID, but a device shared among multiple users via some form of login mechanism creates a separate EID for each user account. Unlike cryptographic host identifiers in SFS and HIP, therefore, EIDs are not only stable but *personal*.

Personal EIDs allow multiple users of a shared UIA host to run independent network services on the device. Since each user's services bind to the user's EID rather than to a host-wide IP address, UIA-aware network applications can run exclusively in the context of the user and rely on UIA to provide user-granularity authentica-

tion and access control. When Bob connects his laptop to the HTTP port at the EID to which PC. Alice resolves, he knows he is connecting to *Alice's* personal web server and not that of another user with an account on the same PC. Alice's web server similarly knows that the connection is coming from Bob and not from someone else using laptop, because her name laptop. Bob resolves to an EID specific to Bob's account on his laptop.

3.2 Naming Principles

Each UIA device acts as an ad hoc name server to support name lookups and synchronize namespace state across devices. UIA names follow the same formatting rules as DNS names, consisting of a series of *labels* separated by dots, and devices resolve UIA names one label at a time from right to left. To resolve the name PC.Alice, for example, Bob's laptop first resolves the rightmost component Alice to find Alice's personal group, and from there resolves the second component PC to find the EID for Alice's PC as named in Alice's personal group.

Whereas DNS resolution traverses a strictly hierarchical tree of "zones" starting from a centrally-managed global root zone, each UIA device has a unique root for resolving UIA names, and users can link UIA groups to form arbitrary graphs. After Bob meets Alice at Time 3 in Figure 2, for example, Bob's "root" group for UIA name resolution, corresponding to his personal group, appears to Alice as a "sub-group" named Bob. Conversely, Alice's "root" group appears to Bob as a "sub-group" named Alice. Since Bob's and Alice's naming relationship forms a cycle in the graph of UIA groups, Bob could for example refer to his own phone via the redundant name phone. Bob. Alice.

UIA groups may at times contain *label conflicts*, or bindings of a single name to multiple distinct targets. When Bob at Time 4 merges his new cell phone with its default name phone into his personal group, which already contains another device named phone, the two phone bindings result in a label conflict. Label conflicts also arise if an ownership dispute splits the *target* that a group name refers to, as described in Section 2.9. Name resolution fails if it encounters a label conflict, preventing the user from following ambiguous links before resolving the conflict. A conflict on one label does not affect the usability of other labels in the same group, however.

3.3 State Management

UIA uses optimistic replication [26, 28, 47] to maintain a user's personal UIA namespace across multiple devices, guarding namespace state against device loss or failure and keeping the namespace available on all devices during periods of disconnection or network partitions. Each device stores in an append-only log all persistent naming state for its user's personal group and any other groups of



Type	Type-specific Record Content
Create	Owner: endpoint ID (EID) of owner device Nonce: ensures uniqueness of new series ID
Link	Label: human-readable string Target: device (EID) or group (series ID) OwnerFlag: grants group ownership if true
Merge Target: series ID (SID) to merge with	
Cancel	Target: record ID to cancel

Figure 4: Log Record Format

interest to the user, and uses an epidemic protocol [12] to distribute updates of each group's state among the devices interested in that group.

UIA's epidemic protocol uses a classic two-phase "push/pull" algorithm. In the "push" phase, when a device creates a new log record or obtains a previously unknown one from another device, it repeatedly pushes the new record to a randomly-chosen peer until it contacts a peer that already has the record. This *rumor mongering* technique works well when few devices have the record, propagating the "rumor" aggressively until it is no longer "hot." In the "pull" phase, each device periodically contacts a randomly-chosen peer to obtain any records it is missing. These *anti-entropy* exchanges work best when most devices already have a record, complementing the rumor mongering phase and ensuring that every device reliably obtains all available records.

4 Naming and Group Management

This section describes in detail how UIA devices manage and synchronize the namespace state comprising their users' personal and shared groups.

4.1 Device Log Structure

UIA organizes the records comprising a device's log into *series*, each series representing the sequence of changes a particular device writes to a particular group. The state defining a group consists of one or more series, one for each device that has written to the group. All devices participating in a group gossip and replicate all records in each of the group's series, preserving the order of records in a given series, but do not enforce any order between records in different series. Since UIA separates the naming state for each group by series, devices can limit gossip to the records relating to groups they're interested in, instead of scanning their neighbors' entire device logs.

As shown in Figure 4, each log record contains a series ID, a sequence number, data specific to the record type, and a signature. The series ID (SID) uniquely identifies the series to which the record belongs. The sequence number orders records within a series. The device that owns a series signs each record in that series with its private key, so that other devices can authenticate copies of records they receive indirectly. A cryptographic hash of the record yields a *Record ID*, which uniquely identifies the record for various purposes described later.

UIA currently defines four record types, listed in Figure 4 and summarized briefly below:

- Create: A create record initiates a new series owned by the device writing the record, as identified in the record's owner field. The owner EID fixes the public/private key pair other devices use to authenticate records in the new series. The record ID of the create record becomes the new series ID; a random nonce ensures the new SID's cryptographic uniqueness. The create record itself is not part of the new series: its own series ID field is usually empty to indicate that it is not part of any series, but it can be non-empty for revocation purposes as described later.
- Link: A link record binds a human-readable label such as Alice to an endpoint ID or series ID denoting the link's target. Links to devices, such as Bob's names laptop and phone, contain the EID of the target device. Links to groups, such as Alice and PhotoClub, contain the SID of some series in the target group. A link record has an owner flag indicating whether the link grants ownership to the link's target, allowing the target to write changes to the group containing the link record. We refer to a link with its owner flag set as a link-owner record.
- Merge: A merge record joins two series to form a single UIA group. The union of all link and cancel records in all merged series determines the set of names that appear in the group, forming a common distributed namespace. A merge takes effect only if the device that wrote the merge record also owns the target group, or if there is a corresponding merge record in the target group pointing back to the first group.
- Cancel: A cancel record nullifies the effect of a specific previous record, specified by the target's record ID. With certain restrictions described below, link records can be canceled to delete or rename group members. Create, merge, and cancel records cannot be canceled.

4.2 Namespace Operations

This section describes how UIA devices implement the important user-visible namespace control operations, in terms of the specific records the devices write to their logs

at the events in the example scenario from Figure 2. The following section will then explain how devices evaluate the contents of their logs to determine the effective state of each group at any point in time.

Device Initialization: When Bob and Alice install or first start UIA on a device at Time 1, the device first writes a create record to its log, forming a new series to represent the user's personal "root" group on that device. The device then writes a link record to the new series, giving itself a suitable default name such as laptop. The device sets the owner flag in this link record to make itself the sole initial owner of the group.

Merging Device Groups: When Bob introduces and merges his devices at Time 2 to form a personal group, each device writes to its own root series a merge record pointing to the other device's root series. These cross-referencing merge records result in a *merge relation-ship* between the two devices, which begin to gossip the records comprising both series so that each device eventually holds a complete copy of each. This merging process does not actually create any new link records, but causes each device to obtain copies of the other device's existing link records (the laptop's link record for its default name laptop and the phone's record for its name phone) and incorporate those names into its own root group.

Aside from merging devices' root series via introduction, a user can use a single device to merge two arbitrary groups, provided the same device already has ownership of both groups. If Bob creates two shared sub-groups and later decides they should be combined, for example, he can merge them on any of his devices. The device writes cross-referencing merge records to the relevant series, exactly as in the introduction scenario.

Meeting Other Users: When Bob and Alice introduce their devices to each other at Time 3, the devices exchange the series IDs of their respective root series, and each device writes a link record to its own root series referring to the other device's root series. Bob's new link record named Alice gives Alice a name in his personal group, and Alice's new link record named Bob likewise gives Bob a name in her group. The devices do not set the owner flags in these new link records, giving Alice and Bob only read-only access to each others' namespaces.

Transitive Merge: Individual merge relationships in UIA are always pairwise, between exactly two series, but merge relationships combine transitively to determine effective group membership. When Bob introduces his cell phone to his home phone at Time 4, the two devices form a merge relationship between their respective root series. Since Bob's home phone and laptop already have a merge relationship, Bob's laptop and cell phone transitively learn about each other via gossiped records they receive from the home phone, and the union of the

records in the three root series determine the contents of the resulting group. Since the merged group has two link records named phone with different target EIDs, the devices flag a label conflict on phone and refuse to resolve this name.

Renaming Labels and Resolving Conflicts: When Bob renames his cell phone to cell at Time 5 to resolve the conflict, his device writes to its root series a cancel record containing the record ID of the link record defining the cell phone's previous name, then writes a new link named cell that is otherwise identical to the original link. Since one of the two conflicting link records is now canceled, the label conflict disappears, and the names phone and cell become usable on all of Bob's devices once they receive the new records via gossip. Bob can resolve the conflict on any of his devices, because any group owner can cancel a link written by another device.

The user can also delete a name from a group outright, in which case the device writes a cancel record without a new link. The ownership granted by a link-owner record, however, can only be nullified by the revocation process described later in Section 4.3.1.

Because UIA implements renames non-atomically with a cancel record coupled with a new link record, if Bob renames Alice to Alicel on his laptop and renames Alice to Alicel on his phone while the two devices are temporarily partitioned, on reconnection he will have two names Alicel and Alicel with no conflict detected. This corner-case behavior, while perhaps slightly surprising, seems acceptable since it "loses" no information and at worst requires Bob to delete one of the resulting redundant names.

Creating Groups: Bob uses his laptop at Time 6 to create his shared PhotoClub group. To create the group, the laptop first writes a create record to generate a fresh series ID. The laptop then writes two link records: first, a link named PhotoClub in its root series pointing to the new series, and second, a link named Bob in the new series pointing back to the root series. The laptop sets the owner flag in only the latter link record, giving Bob's personal group ownership of the new group, without giving PhotoGroup ownership of Bob's personal group.

Suppose that Bob now uses a different device, his cell phone for example, to add Alice to PhotoClub. Bob's cell phone is already an indirect owner of PhotoClub, because the cell phone is an owner of Bob's personal group and Bob's personal group owns PhotoClub. The cell phone does not yet have a series in PhotoClub, however, to which it can write records: initially only the laptop, which created the new group, has a series in the group, and only it can sign records into that series. The cell phone therefore creates its own PhotoClub series, by writing a create record to form a new series owned

by itself, and then writing a merge record to this new series pointing to the laptop's PhotoClub series. Although no corresponding merge record in the laptop's PhotoClub series points back to the cell phone's new series (in fact the laptop may be offline and unable to sign such a record), the cell phone's merge record takes effect "unilaterally" by virtue of the cell phone's indirect ownership of PhotoClub. The cell phone then writes a copy of Bob's link to Alice into its new PhotoClub series, and other devices learn of the new series and the new name as they gossip records for PhotoClub.

Revoking Ownership: When Bob learns at Time 7 that his cell phone is missing, he uses his laptop to revoke the cell phone's ownership of his personal group, either by deleting the name cell from his personal group or by clearing its owner flag. To implement this revocation, however, Bob's laptop cannot merely write a cancel record pointing to the link record for cell: the cell phone would still own a series in Bob's personal group and thus retain "hidden" control over the group.

To revoke the cell phone's ownership, therefore, Bob's laptop creates a new personal group for Bob and copies the original group's name content into it. To create the new group, the laptop writes a create record whose series ID field is not empty as usual, but instead contains the SID of the laptop's original root series. The laptop then writes link records to the new series corresponding to all the active links in the old series, omitting links or ownership flags to be revoked. The create record written into the old root series indicates to all interested devices that the new series forms a group that is intended to replace or act as a *successor* to the original group.

As long as only one such "create successor" record exists in Bob's old personal group, all devices treat links to any series in the old group as if they linked to the successor group instead. Upon receiving via gossip the records describing Bob's new group, for example, Alice's devices subsequently resolve her name Bob to the new group, and use it to calculate which devices should be given access to resources she has authorized Bob to access, effectively revoking the cell phone's access.

If the cell phone writes a conflicting "create successor" record to *its* series in Bob's original group, however, then the original group becomes *disputed*, and other devices refuse to resolve links to any series in the original group as soon as they learn about the dispute. Alice's devices thus refuse to resolve her name Bob and deny access to any resources she authorized using that name. Once Alice updates her broken link to refer to the correct successor group, either by re-introducing with Bob or by copying a fresh link from a mutual friend, her device writes a new link referring to a series in Bob's new group, the old group becomes irrelevant and Bob can again access Alice's resources via the devices in his new personal group.

```
global M: membership table: SID \rightarrow SID set
global O: ownership table: SID set \rightarrow EID set
function eval_membership_ownership():
  for each known series sid:
     M[sid] \leftarrow \{sid\}
    O[\{sid\}] \leftarrow EID of device that owns series sid
     for each link-owner record in each series sid:
       if link target is a device teid:
         O[M[sid]] \leftarrow O[M[sid]] \cup teid
       else if target is a series tsid:
         O[M[sid]] \leftarrow O[M[sid]] \cup O[M[tsid]]
     for each merge record in each series sid:
       tsid \leftarrow target series ID of merge record
       O[M[sid]] \leftarrow O[M[sid]] \cup O[M[tsid]]
       if owner EID of series sid \in O[M[tsid]]:
         O[M[sid] \cup M[tsid]] \leftarrow O[M[sid]] \cup O[M[tsid]]
         for each series ID msid \in M[sid] \cup M[tsid]:
            M[msid] \leftarrow M[sid] \cup M[tsid]
  until M and O stop changing
```

Figure 5: Membership and ownership evaluation pseudocode

If link or cancel records exist on Bob's other devices that his laptop has not yet received at the time of revocation, the laptop cannot copy these change records into the new group and they become *orphaned*. Bob's devices continue to monitor and gossip records in the old group after the revocation, however, to detect both orphans and ownership disputes. If a device with ownership of the new group detects an orphaned record written by itself or another device with ownership of the new group (not a revokee), it automatically "forwards" the change by writing a corresponding record to the new group.

4.3 Group State Evaluation

This section describes the algorithms UIA devices use to determine the current state of a given group from the set of log records they have on hand. Devices evaluate group state in three stages: (1) membership and ownership, (2) group successorship, and (3) name content.

4.3.1 Membership and Ownership

In the first stage, a UIA device collects the series IDs referred to by all records in its log, and clusters them into sets based on merge relationships to form UIA groups. At the same time, the device computes the set of device EIDs to be considered owners of each group, either directly or transitively. Group membership and ownership must be computed at the same time because they are mutually dependent: group membership expansion via merge can introduce additional owners, and owner set expansion can place additional merge records under consideration.

Figure 5 shows pseudocode for membership and ownership evaluation. The algorithm uses a membership table M mapping each known series ID to a set of series IDs

sharing a group, and an ownership table ${\cal O}$ mapping each group (represented by a set of series IDs) to a set of owner device EIDs. The algorithm first initializes the entry in ${\cal M}$ for each series to a singleton set containing only that series, and initializes the owner set entry in ${\cal O}$ for each such singleton group to the EID of the device that owns that series. The algorithm then repeatedly merges groups and expands ownership sets until it reaches a fixed point. The algorithm terminates because member and owner sets only grow, and each device knows of a finite number of series IDs at a given time.

In each iteration, the algorithm first follows link-owner records, expanding the ownership set of the group containing a link-owner record according to the target device EID or the current ownership set of the target group, as applicable. Across iterations, this step handles transitive propagation of ownership across multiple groups, such as Bob's laptop's ownership of PhotoClub via the laptop's ownership of Bob's personal group.

Second, for each merge record, the algorithm expands the ownership set of the group containing the merge record to include the ownership set of the target group, then checks whether the device that wrote the merge record is *authorized* by virtue of having ownership of the target group. The authorization check prevents a device from merging a series into an arbitrary group without permission. In the symmetric case where two merge records refer to each others' series IDs, each merge is authorized by the fact that the other merge grants ownership of its own series to its target. Once a merge is authorized, the algorithm combines the SID sets of the respective groups to form one group containing all the merged SIDs, and similarly combines the respective owner sets.

4.3.2 Group Successorship

In the second stage, a device computes the *successorship* status of each group resulting from the first stage, in order to handle revocations and ownership disputes. The device first forms a directed graph reflecting immediate successor relationships: a create record in series A yielding a new series B makes the group containing B a successor to the group containing A. Next, the device takes the transitive closure of this graph to form a transitive successorship relation: if B succeeds A and C succeeds B, then C transitively succeeds A.

The device now assigns to every group G one of three states as follows. If G has no successors, it is a *head* group: no revocations have been performed in the group, and links to series IDs in the group resolve normally. On the other hand, if there is a second group G' that is a transitive successor to G and is also a transitive successor to all other transitive successors to G, then G' is the *undisputed successor* to G. In this case, links to series IDs in group G resolve to group G' instead. Finally, if G has

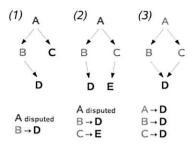


Figure 6: Example Group Successorship Scenarios

successors but no undisputed successor, then group G is disputed, and links to series IDs in G do not resolve at all.

Figure 6 illustrates several group successorship scenarios and the corresponding results of this algorithm. In scenario (1), two conflicting revocations have placed group A under dispute; A's successor B also has a successor due a second revocation in B but B is not under dispute. Scenario (2) is like (1) except a revocation has also been performed in group C, forming a new head group E. Scenario (3) shows the result after the warring owners in (2) settle their differences and merge their head groups D and E, resolving the original dispute over group A.

4.3.3 Name Content

In the third and final stage, for each head group to be used for name resolution, a device computes the group's namespace state as follows. Given the set of all link records in every series in the group, the device removes all link records targeted by a cancel record in any series of the group to form the set of *active* links. Any device that owns a group can cancel a link written by another device, but a cancel cannot revoke ownership.

The set of *active labels* in a group, shown in a namespace browser for example, is the set of labels appearing in any active link record in the group. To be *usable*, all active links for a given label must have the same permissions, and must target the same device EID or SIDs in the same group. Otherwise the label is *in conflict*, as Bob's home and cell phone are at Time 4 in the example. If Bob creates identical links on different devices independently, such as by separately introducing both his cell phone and his laptop to Alice to yield duplicate Alice links, this action does not create a label conflict when Bob merges his home and cell phone together because the redundant links have the same target and permissions.

5 Routing and Forwarding

Once the naming layer has resolved a device name to a location-independent EID, UIA's routing layer is responsible for *locating* the target device—finding its current IP address—and *forwarding* traffic to it through other devices if direct connectivity is unavailable.

Many designs are conceivable that would perform these two functions, such as a general-purpose overlay routing algorithm we explored previously [16]. In this work, however, we adopt a simple design that does not attempt to provide connectivity between arbitrary devices, but is optimized for connecting to devices in the user's immediate *social neighborhood*: primarily the user's own devices and those of friends named in the user's personal group, and occasionally "friends of friends," but rarely more indirect contacts. In practice we expect users to create (or copy from other users) names in their own personal groups for others with whom they wish to interact regularly, justifying our assumed usage model.

In brief, UIA builds an overlay network between devices in its social neighborhood. To locate a remote device by its EID, a device floods a *location* request through the overlay to discover the EIDs, IP addresses, and ports of devices forming a path through the overlay to the target. The originating device then connects directly to the target's discovered IP address and port, or if the target is not directly reachable (e.g., because of an intervening NAT), *forwards* traffic to it by source-routing data via the existing connections in the discovered path.

5.1 Overlay Construction and Maintenance

Each UIA device maintains an open TCP connection with up to a configurable number of overlay *peers*. Ideally, these peers should be on the public Internet, so that a device behind a NAT can receive messages from devices outside via its active peering connections. A device should choose other devices when none on the public Internet are reachable, however, so that the overlay remains useful in ad hoc environments. Furthermore, the devices of friends should be close to each other in the overlay, so that location or forwarding paths between them are short.

To meet these goals, a device first prefers as peers devices that are *stable*, and secondarily prefers those that are closest to it in *friendship distance*. A device is considered *stable* if it does not have a private IP address [41] and has met a threshold level of availability in the recent past. A peer's *friendship distance* is roughly the number of labels in the local device's shortest name for that peer. The rest of this section explains how a device discovers stable peers and calculates friendship distances.

Each device maintains a *potential peer set* that contains potential peers' EIDs and the times, IP addresses, and ports at which the device has connected to those peers in the past. Initially, a device populates this set with the devices to which the user has directly introduced the device. To discover new potential peers, a device periodically exchanges its potential peer set with those of other devices within a configurable maximum friendship distance. A device adds to the set only those devices to which it is able to establish a TCP connection when it discovers them.

A device classifies a potential peer as *stable* if it meets an availability threshold (e.g., 90%) at the same public IP address and port in the recent past (e.g., the last week). To monitor availability, a device periodically chooses a random potential peer and attempts a connection to its last known location. A device need not have a static IP address to be classified as stable: a device with a dynamic non-private IP address that changes infrequently, such as a home PC left on and connected via a DSL or cable modem, will also typically be classified as stable.

A device computes the *friendship distance* of each of its potential peers by assigning a distance of 1 to its *direct peers*: those the naming layer identifies as devices in the user's personal group and in groups to which the user has linked (the user's immediate friends). The device then assigns distances to indirect peers transitively, giving the direct peer of a direct peer a distance of 2, for example.

To improve robustness, a device manufacturer can seed the potential peer sets of its products with a set of *default peers*, which devices treat as having an infinite friendship distance. Two newly-purchased mobile devices, after being introduced and exchanging potential peer sets, thus have at least one stable peer in common at the outset to help them re-connect after a move. Once the mobile devices discover other stable peers at smaller friendship distances, however, they prefer the new devices over the default peers, mitigating the manufacturer's cost in providing this robustness-enhancing service.

5.2 Token-limited Flooding

To communicate with a remote device, a device first attempts a direct TCP connection to the IP address and port at which it last connected to the target, if any. If this connection fails or the originator has no address information for the target device, it floods a location request through the overlay to locate the target by its EID.

UIA uses a *token count*, in place of the traditional hop count [6], to limit the scope of location request floods. The token count bounds the total number of *devices* to which a request may be forwarded, rather than the number of times each request may be re-broadcast. This distinction is important for two reasons. First, although devices seek to connect with a fixed number of peers, the number of devices that choose a given device depends on the target's stability and popularity, so the overlay's degree is highly non-uniform. Hop count is thus a poor predictor of the number of devices a request will reach. Second, the overlay network is highly redundant: two friends' devices are likely to share many common peers, for example, so searching *all* devices within some distance of a request's source is often unnecessary.

Location requests contain the EIDs, IP addresses, and ports of devices they have traversed; devices forward responses back through the overlay along the same path.

A device with an open TCP connection to a request's target immediately responds with the target's IP address and port. Otherwise, it subtracts one token for itself, divides the other tokens among its peers not already in the path, distributing any remainder randomly, and forwards the request to those peers that receive a non-zero count. The device retains the request's target EID and return path for a short period, waiting for the forwarded requests to complete, and replying to the original request when any of the forwarded ones succeed or when all of them have failed. A request also fails if the source has not received a successful response within a timeout. If a device receives a duplicate request for the same EID as an outstanding request (e.g., along a different path), it forwards the new request anyway according to its token count, giving peers for which there were not enough tokens in previous instances another chance to receive the request.

As shown in Section 7, most location requests succeed within the near vicinity of the source in the overlay network. To limit the cost of the search, a device thus initially sends each request with a limited number of tokens and retries after each failure with a multiplicatively increased number, up to some maximum.

5.3 Source-Routed Forwarding

To communicate with the target device after receiving a successful location response, the originator tries to open a direct connection to each device in the response path, starting with the target itself and proceeding backwards along the path until a connection succeeds. The originator then source-routes messages to the target along the tail of the path starting with the device to which it connected.

Consider for example two devices a and b behind different NATs, both of which peer with a common stable device s. When a performs a location request for b's EID, it discovers the path $a \rightarrow s \rightarrow b$. Device a then tries to open a direct connection to b, but b's NAT blocks that connection, so a forwards traffic to b through s instead. Device s itself initiates no location requests, but merely forwards traffic along the path specified by a.

6 Implementation

A prototype UIA implementation currently runs on Linux and Mac OS X. As illustrated in Figure 7, the prototype consists of two user-level daemons implementing UIA's naming and routing layers, respectively, and a graphical application for browsing and controlling devices and groups. The control application and other UIA-aware applications on the device interface directly to the naming and routing daemons via Sun RPC. Through these interfaces, UIA-aware applications can resolve UIA names to EIDs, explore and modify groups on behalf of the user, send packets to EIDs, receive packets on the device's EID, and discover peers on the local-area network.

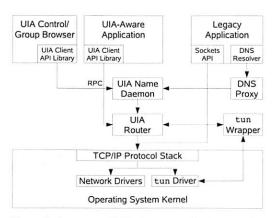


Figure 7: Structure of UIA Prototype Implementation

6.1 Prototype Status

The name daemon is written in Python and implements the design described in Section 4, providing group creation, merging, named links between groups, naming state gossip, state evaluation, multi-component name resolution, and ownership revocation. The name daemon does not yet detect and copy orphaned change records across revocations as described in Section 4.2, however.

The prototype routing daemon implements in C++ the algorithms described in Section 5. The router uses Bonjour for local-area device discovery, and uses SSL over TCP connections for secure communication.

The UIA control application allows the user to browse the UIA namespace and create and modify groups, as illustrated earlier in Figure 3, and supervises the device introduction process as illustrated in Figure 1. The control application is still unpolished and does not yet fully support shared groups or revocation, however.

6.2 Support for Smaller Devices

We have ported the UIA prototype to the Nokia 770 Internet Tablet, a Linux-based Internet appliance with an ARM processor. The naming and routing layers have the full functionality of the regular Linux/Mac version of UIA, but the port of the GUI control application is not yet complete. In general, we expect the routing and naming modules to port easily among smaller devices, while the GUI component requires more modifications because of the more specialized and restrictive user interface frameworks available on each class of mobile device. UIA does not rely on extensive data entry or other forms of user interaction that are fundamentally difficult to achieve on small devices, however.

6.3 Legacy Application Support

The UIA prototype supports legacy applications through a tun wrapper and DNS proxy. The tun wrapper disguises EIDs as device-local IP addresses and uses the kernel's tun device to forward applications' TCP and UDP packets for these special IP addresses over UIA's routing layer. The DNS proxy similarly intercepts name lookups made by local applications and resolves UIA names to device-local IP addresses for the corresponding EIDs. We have run Apache, Firefox, OpenSSH, and Apple's Personal File Sharing over UIA using this legacy interface without modification or recompilation.

UIA's legacy application support layer makes the user's personal group appear to applications like a global virtual private network, by intercepting network-local broadcast packets that applications send to UIA's special IP addresses and forwarding them securely to each of the user's personal devices. Because of this feature, many broadcast-based "local-area" service discovery protocols such as Bonjour automatically work across all the devices in the user's personal group, even when some of the devices are in fact remote. We have used Apple's Bonjour-based Personal File Sharing, for example, to locate and share files remotely between devices in a UIA personal group as if they were present on the same LAN.

6.4 Experience with UIA

We currently run the UIA prototype on a number of desktop and laptop machines in our lab, and regularly run existing applications such as SSH over UIA to reach our mobile devices via their short personal device names. UIA automatically accounts for IP address changes and traverses NATs as necessary; SSH connections open when we take a laptop home need not be restarted. Although these uses are already possible via alternate protocols such as mobile IP, the complexity of configuring these alternatives has generally deterred even those of us with the necessary technical knowledge from deploying them. We feel that UIA's zero-configuration paradigm for personal naming and connectivity provides a crucial missing element in making mobile devices usable.

7 Evaluation

UIA's primary goal is convenience and usability by nontechnical users, a goal that can only be evaluated effectively once UIA has been deployed longer and more widely in the field. We can however evaluate key performance characteristics of the routing layer through simulation, to verify that the proposed design is capable of providing the desired connectivity on realistic networks.

7.1 Experimental Setup

We use as our simulated network a crawl of the social networking site Orkut gathered by Li and Dabek [29]. This graph is merely suggestive; until UIA is more widely deployed, it will not be clear how accurately the Orkut graph characterizes UIA's likely usage model. The graph has 2,363 users, which we take to represent devices, as if each user owned one device. Friend relationships are bidirectional, and the number of friends per user is highly

skewed: the median is only 7, but the most popular user has over 1,000.

Our simulator takes as input a *percent stable* parameter and randomly chooses that percent of the devices to be *stable* and publicly accessible. All other devices the simulator considers to be *mobile* and inaccessible except through some stable device, as if behind a NAT. We assume that all devices agree as to which devices are stable.

Each device chooses 16 peers and allows at most 64 devices to choose it, to limit each device's overlay maintenance cost in a real network. Devices choose peers in order of friendship distance. A device can only choose a given target as a peer if the target does not already have 64 peers, or if the new device is closer than one of the target's existing peers, in which case the target discards a random peer at higher friendship distance. Since we do not yet have traces with which to simulate the network's evolution, the simulated devices choose peers in random order, iterating until the network reaches a fixed point.

The simulator then performs token-limited location requests on the resulting overlay between 10,000 random pairs at friendship distances 1, 2, and 3. Each lookup starts with 16 tokens, and doubles after each failure, up to a maximum of 256 tokens. The simulator records the percentage of requests that succeed after a given number of rounds and the total number of messages sent.

7.2 Location Success Rate

An important performance metric for the location algorithm is the number of tokens needed to locate a target device successfully. Using more tokens increases the chance of success (assuming that the overlay is in fact connected), but also increases the cost of concluding that an unreachable device is offline. Figure 8 shows the success rate measured in the simulation for locating devices at friendship distance 1. Using 256 tokens, the location algorithm achieves greater than a 99.5% success rate for 10% or more stable devices. Using 64 tokens the algorithm achieves 97.5% success for 10% or more stable devices. The vast majority of requests—80% of requests at 10% or more stable devices—succeed within the first inexpensive round of 16 tokens.

At the far left of the graph where few stable devices are available, the success rate drops off because each stable device can only support 64 peers, and there are not enough stable devices for each mobile device to choose a full 16 peers, or in some cases any. As the percentage of stable devices increases, a linearly increasing number of location requests are to stable devices that can be contacted directly without flooding, thus requiring no tokens.

We also measured the success rates for locating devices at friendship distances of 2 and 3, though we omit these graphs for space reasons. The results for distance 2 are almost as good as for distance 1, presumably be-

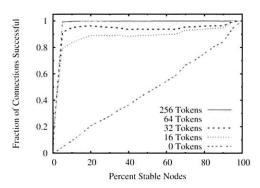


Figure 8: Location request success rate

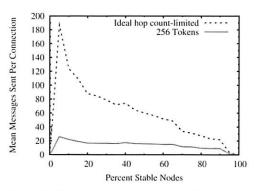


Figure 9: Mean messages sent per location request

cause two devices at friendship distance 2 are likely to peer with some common stable device at distance 1 from each of them. Success rate drops considerably at distance 3, however, achieving only 50% success with 256 tokens in networks of 40% or more stable devices, for example.

7.3 Messages Sent

The lower line in Figure 9 shows the total number of messages sent during successful token-limited lookup requests for devices at friendship distance 1. A request's message count is ultimately bounded by its token count, but is often much less because successful lookups usually do not require all available tokens.

At the left edge of the graph, there are not enough stable devices for every mobile device to have a peer, so few requests succeed. Those that do succeed do so cheaply, however, because all of the connected mobile devices have clustered around the same few stable devices. The message count peaks near the point where the number of stable devices becomes barely sufficient to serve all of the mobile devices, so the requests usually succeed but only after contacting many devices. As the number of stable devices increases further, more requests complete without flooding at all, since stable targets are reachable directly at their last known IP addresses.

To contrast UIA's token-limited scheme with flooding limited by hop count [6], the upper line in the fig-

ure shows the total number of messages sent for successful hop count-controlled location requests in which the originating device knows via an oracle the exact hop count required for the search to succeed in one round. As the graph shows, the token-based scheme requires far fewer messages than even this "ideal" hop count-limited scheme. The inefficiency of the hop count scheme results from the skewed popularity distribution and redundancy of the friendship graph, as discussed in Section 5.2.

8 Future Work

Although we feel that the current UIA prototype demonstrates a promising approach to naming and connecting personal devices, many avenues for future work remain, some of which are highlighted in this section.

8.1 Naming

UIA currently provides no read access control for its namespaces, only write access control via group ownership. Users may wish to hide certain names, such as links to business partners, from view of the general public, or limit visibility of devices at work to business colleagues while allowing family members to see devices at home.

The naming layer currently assumes that groups are small and change infrequently, so that it is reasonable for devices always to gossip entire groups and store change records forever (or until the device is replaced or the user's account wiped). A traditional DNS-like remote name resolution protocol might usefully supplement UIA's gossip protocol, allowing devices to resolve names in large or rarely accessed groups held on other devices without replicating the entire group. A UIA device might also keep a separate log for each group or series, and garbage collect logs of groups the device does not own and has not accessed recently. A state checkpoint mechanism might similarly enable devices to garbage collect old change records for groups they own.

UIA currently assumes that groups are owned by one person or a few people managing the group by consensus: any group owner can modify the group without restriction. Users may wish to configure groups so that changes require approval from multiple distinct owners, or to make some owners more "trusted" than others. Treating a PC locked up in a data center as more trustworthy than a laptop or cell phone could eliminate the risk of ownership disputes if the mobile device is stolen, for example. The user would have to think ahead and perform more manual configuration, however, and the consequences might be worse if the trusted PC is compromised.

As an alternative to the digital signature algorithm with which UIA normally signs namespace change records, we are experimenting with a security framework based on proof-carrying authentication [1]. In this framework, instead of a signature, a change record contains a structured

proof that the record's meaning (e.g., "resolve name x to EID y") has been endorsed by the group owner. Proof-carrying authentication enables new types of proofs to be created and deployed without changing the verifier's code. We have used this mechanism for example to create a UIA group whose records are certified by MIT's central X.509 certification authority (CA), so that alice.mit securely maps to the person the MIT CA has endorsed as alice@mit.edu even though UIA contains no explicit code to check X.509 certificates.

8.2 Routing

The UIA routing layer currently uses forwarding for NAT traversal, which is a general but inefficient solution. As an optimization, we plan to incorporate hole punching [18], a technique that can build direct peer-to-peer connections across many types of NAT without forwarding. Since this and other NAT traversal techniques [8,48] only work with certain NATs, however, UIA will still need forwarding as a fallback to provide robust connectivity.

The routing layer currently uses TCP for all UIA connectivity, including for tunneling the UDP datagrams of legacy applications, limiting the prototype's effectiveness for handling real-time data such as streaming media. We intend to introduce a UDP-based UIA connectivity protocol to provide more effective best-effort delivery.

The routing layer's search algorithm could use additional hints from the naming layer to improve its performance. To locate laptop.Charlie.Bob.Alice, for example, it might first locate some device belonging to Bob and ask that device to locate laptop.Charlie.

8.3 Legacy Application Support

UIA's legacy application interface currently cannot provide each user of a multi-user machine with a fully separate TCP/UDP port space for its own EID, because the kernel's protocol stack offers no way to ensure that only a particular user's applications can bind a socket to the device-local IP address representing that user's EID. Thus, without enhancing the kernel's transport protocols, only UIA-aware applications can make full use of personal EIDs. Fixing this issue requires changes to kernel-level code and is thus less portable.

9 Related Work

UIA builds on a large body of related work in the areas of naming systems, location-independent identifiers, gossip and optimistic replication protocols, and social networks.

UIA's personal naming model is inspired in part by SDSI/SPKI [14, 42]. Like SDSI, UIA allows users to define locally-scoped personal names bound to cryptographic targets and groups to form decentralized, composable namespaces. While SDSI associates public keys with users (principals) and expects users to know about

and manage their own public keys, however, UIA simplifies key management by making each device responsible for creating and managing its own device-specific key invisibly to the user. UIA devices form *user* identities out of cooperating groups of personal devices, which the user builds through simple device introduction and merge.

Existing Internet protocols can provide some of UIA's connectivity features, but they require configuration effort and technical expertise that deters even sophisticated users. Dynamic DNS [49] can name devices with dynamic IP addresses, but requires configuration on both the name server and the named device, and devices still become inaccessible when behind NAT. DNS Security [4] cryptographically authenticates DNS names, but its administration cost has hindered deployment even by the Internet's root naming authorities, let alone by ordinary users. Mobile IP [37] gives a mobile device the illusion of a fixed IP address, but requires setting up a dedicated forwarding server at a static, public IP address. Virtual Private Networks (VPNs) [22] provide secure remote access to corporate networks, but their infrastructure and administration requirements make them unsuitable for deployment by average consumers for their personal networks.

Uniform Communication Identifiers [15] provide a common identifier for phone, E-mail, and other forms of communication, along with a central address book shareable among communication devices. HINTS [30] uses name-history trails to map obsolete user names to current ones. These systems still rely on globally unique names with centralized registration and management, however.

Bonjour [2] allows devices to choose their own names on local-area networks, but these names are insecure and ephemeral: any device can claim any name, and its name becomes invalid as soon as it moves to a different network. UIA uses Bonjour libraries to discover new devices on the local network, but UIA names persist and remain bound to the original target device despite later migration.

UIA builds on host identity ideas developed in SFS [32], HIP [34], JXTA [23], and i3 [45], introducing cryptographic EIDs that securely identify not just a host but a particular *user* on that host. Different users of a shared UIA host can run independent personal network services without conflicting or requiring host-wide configuration, and network services can leverage UIA names and EIDs to authenticate clients at user granularity.

Distributed hash tables (DHTs) [5,43,44] provide scalable lookup of arbitrary flat identifiers in large distributed address spaces, but tolerate only limited asymmetry or non-transitivity in the underlying network [20]. UIA's router in contrast handles asymmetries such as those caused by NATs, but does not attempt to resolve *arbitrary* identifiers reliably: UIA instead focuses on reliable routing to devices nearby in the user's social network, for which scoped flooding [6] is more suitable.

DHT-based naming systems such as DDNS [9], i3 [45], and CoDoNS [40] provide new mechanisms for resolving global names. TRIAD [7] provides content delivery and NAT traversal by routing on global DNS names. In place of global names, UIA focuses on global connectivity via personal names, which users can choose without the restriction of global uniqueness. In addition, UIA's optimistic replication of naming state keeps the user's namespace available on his devices even while disconnected from the Internet and its global name services.

Ficus [24, 26], Coda [28], and Ivy [35] develop optimistic replication algorithms for file systems, and Bayou [47] does so for databases. Rumor [25] and P-Grid [11] explore optimistic data replication on mobile devices, Roma [46] uses one mobile device to offer central management of data on other devices, and Footloose [36] uses mobile devices the user carries to propagate updates among other devices. UIA builds on all of this work to address distributed naming and ad hoc group management, confronting the additional challenge of maintaining consistency when not only the *data content* but the *set of participants* may change independently on different devices.

UIA is a continuation of work begun with Unmanaged Internet Protocol [16, 17]. UIA extends the earlier work with its personal naming system, and by leveraging the user's social network for routing purposes as in sybil-resistant DHTs [10] and social data sharing systems such as Turtle [38], SPROUT [31], F2F [29], and Tribler [39].

10 Conclusion

This paper proposes the Unmanaged Internet Architecture for introducing, naming, and globally connecting mobile devices. UIA gives users persistent personal names for conveniently finding and expressing who they want to talk to, what devices they wish to access, and who can access their own devices.

Each device starts with a generic name for itself, such as laptop, and a cryptographic end-system identifier to provide authentic and private communication. A user can merge devices to form personal groups, which cooperate to maintain a distributed namespace by gossiping logs of the user's changes. A user's group can name both the user's devices and other users' groups; users can form links securely either by physical device introduction or via other trusted channels. Since UIA names are local and personal, users need not register with central authorities to obtain scarce globally unique names.

UIA uses ad hoc routing through social neighbors' devices to cope with a spectrum of connectivity environments. Scoped flooding ensures robustness when groups of devices form isolated islands of connectivity, and a social overlay enables devices to find a target's current IP address efficiently when they have Internet connectivity.

Acknowledgments

This research is sponsored by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan, and by the National Science Foundation under Cooperative Agreement ANI-0225660 (Project IRIS). We would like to thank Martin Abadi, Tom Rodeheffer, Nokia Research Center Cambridge, and the USENIX reviewers for support and feedback on early paper drafts.

References

- Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In 6th ACM CCS, November 1999.
- [2] Apple Computer, Inc. Bonjour. http://developer.apple.com/networking/bonjour/.
- [3] Apple Computer, Inc. FileVault. http://www.apple.com/macosx/features/filevault/.
- [4] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements, March 2005. RFC 4033.
- [5] Hari Balakrishnan et al. Looking up data in P2P systems. Communications of the ACM, February 2003.
- [6] Yatin Chawathe et al. Making Gnutella-like P2P systems scalable. In ACM SIGCOMM, pages 407–418, August 2003.
- [7] David R. Cheriton and Mark Gritter. TRIAD: A new next-generation Internet architecture, July 2000.
- [8] Stuart Cheshire, Marc Krochmal, and Kiren Sekar. NAT port mapping protocol, June 2005. Internet-Draft (Work in Progress).
- [9] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS using Chord. In *1st IPTPS*, March 2002.
- [10] G. Danezis, C. Lesniewski-Laas, F. Kaashoek, and R. Anderson. Sybil-resistant DHT routing. In ESORICS, 2005.
- [11] A. Datta, M. Hauswirth, and K. Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In 23rd ICDCS, 2003.
- [12] Alan Demers et al. Epidemic algorithms for replicated database maintenance. In 6th PODC, pages 1–12, New York, NY, 1987.
- [13] Steve Dohrmann and Carl Ellison. Public-key support for collaborative groups. In *1st Annual PKI Research Workshop*, April 2002.
- [14] C. Ellison et al. SPKI Certificate Theory, 1999. RFC 2693.
- [15] European Telecommunications Standards Institute. User identification solutions in converging networks, April 2001.
- [16] Bryan Ford. Scalable Internet routing on topology-independent node identities. Technical Report 926, MIT LCS, October 2003.
- [17] Bryan Ford. Unmanaged Internet protocol: Taming the edge network management crisis. In *HotNets-II*, 2003.
- [18] Bryan Ford. Peer-to-peer communication across network address translators. In USENIX, Anaheim, CA, April 2005.
- [19] Bryan Ford et al. User-Relative Names for Globally Connected Personal Devices. In 5th IPTPS, February 2006.
- [20] Michael J. Freedman et al. Non-transitive connectivity and DHTs. In USENIX WORLDS 2005, December 2005.
- [21] Cyril Gavoille and Marc Gengler. Space-efficiency for routing schemes of stretch factor three. JPDC, 61(5):679–687, 2001.
- [22] B. Gleeson et al. A Framework for IP Based Virtual Private Networks, February 2000. RFC 2764.
- [23] Li Gong. JXTA: A network programming environment. IEEE Internet Computing, 5(3):88–95, May 2001.
- [24] R. G. Guy, G. J. Popek, and T. W. Page, Jr. Consistency algorithms for optimisic replication. In First International Conference on Network Protocols, 1993.
- [25] Richard Guy et al. Rumor: Mobile data access through optimistic peer-to-peer replication. In ER Workshops, pages 254–265, 1998.
- [26] Richard G. Guy et al. Implementation of the Ficus replicated file system. In USENIX Summer Conference, pages 63–71, June 1990.
- [27] M. Holdrege and P. Srisuresh. Protocol complications with the IP network address translator, January 2001. RFC 3027.

- [28] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In 13th SOSP, pages 213–225, 1991.
- [29] Jinyang Li and Frank Dabek. F2F: Reliable storage in open networks. In 5th IPTPS, Santa Barbara, CA, February 2006.
- [30] Petros Maniatis and Mary Baker. A historic name-trail service. In 5th WMCSA, October 2003.
- [31] Sergio Marti, Prasanna Ganesan, and Hector Garcia-Molina. SPROUT: P2P routing with social networks. In P2P&DB, 2004.
- [32] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In SOSP, 1999.
- [33] P. Mockapetris. Domain names: concepts and facilities, November 1987. RFC 1034.
- [34] R. Moskowitz and P. Nikander. Host identity protocol architecture, April 2003. Internet-Draft (Work in Progress).
- [35] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In 5th OSDI, 2002.
- [36] J.M. Paluska et al. Footloose: A case for physical eventual consistency and selective conflict resolution. In 5th WMCSA, 2003.
- [37] C. Perkins, Editor. IP mobility support for IPv4, August 2002. RFC 3344.
- [38] B. Popescu, B. Crispo, and A. Tanenbaum. Safe and private data sharing with Turtle. In 12th Workshop on Security Protocols, 2004
- [39] J.A. Pouwelse et al. Tribler: A social-based peer-to-peer system. In 5th IPTPS. February 2006.
- [40] Venugopalan Ramasubramanian and Emin Gün Sirer. The design and implementation of a next generation name service for the Internet. In ACM SIGCOMM, August 2004.
- [41] Y. Rekhter et al. Address allocation for private internets, February 1996. RFC 1918.
- [42] R.L. Rivest and B. Lampson. SDSI: A simple distributed security infrastructure, April 1996. http://theory.lcs.mit.edu/~cis/sdsi.html.
- [43] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.
- [44] Ion Stoica et al. Chord: A scalable peer-to-peer lookup service for Internet applications. In SIGCOMM, 2001.
- [45] Ion Stoica et al. Internet indirection infrastructure. In SIGCOMM, August 2002.
- [46] E. Swierk, E. Kıcıman, V. Laviano, and M. Baker. The Roma personal metadata service. In 3rd WMCSA, December 2000.
- [47] Douglas B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In 15th SOSP, 1995.
- [48] UPnP Forum. Internet gateway device (IGD) standardized device control protocol, November 2001. http://www.upnp.org/.
- [49] P. Vixie, Editor, S. Thomson, Y. Rekhter, and J. Bound. Dynamic updates in the domain name system, April 1997. RFC 2136.

A Modular Network Layer for Sensornets

Cheng Tien Ee*, Rodrigo Fonseca*, Sukun Kim*, Daekyeong Moon*, Arsalan Tavakoli*, David Culler*†, Scott Shenker*‡, Ion Stoica*

Abstract

An overall sensornet architecture would help tame the increasingly complex structure of wireless sensornet software and help foster greater interoperability between different codebases. A previous step in this direction is the Sensornet Protocol (SP), a unifying link-abstraction layer. This paper takes the natural next step by proposing a modular network-layer for sensornets that sits atop SP. This modularity eases implementation of new protocols by increasing code reuse, and enables co-existing protocols to share and reduce code and resources consumed at run-time. We demonstrate how current protocols can be decomposed into this modular structure and show that the costs, in performance and code footprint, are minimal relative to their monolithic counterparts.

1 Introduction

The field of wireless sensornets (hereafter, sensornets) has made great strides over the past few years, producing better devices, larger deployments, and more functional and stable systems. The different and varied nature of sensornet applications, coupled with heavy need for optimization, called for an exploratory phase in which boundaries between hardware/software, application/OS, and networking components were flexible and in flux [7]. As a result, there are several vertically integrated designs, created by separate research groups, which employ quite different modularities.

Across these designs, there is a general lack of consistency in terms of the functionalities implemented in modules as well as their interfaces, resulting in unnecessary coupling between modules. The creation of new protocols thus requires more effort to reorganize functionalities or even reimplement them from scratch. ¹ Inconsistencies in service interfaces also cause the porting

of applications onto different protocols to become nontrivial. Additionally, co-existing protocols with modules implementing overlapping functionalities unnecessarily consume more resources in terms of memory and energy, and can therefore reduce the lifetime of a sensornet. Thus, the lack of an overall sensornet architecture minimizes code reuse, complicates porting, and leads to increased memory consumption for already resourceconstrained systems.

The first steps towards such an architecture were taken in [4, 21], which identified the narrow waist of the sensornet architecture as lying between the link and network layers. In this paper we take the next step in this endeavor by defining a modular network layer.

Our goals are simple: to increase code reuse and runtime sharing. Code reuse will foster more rapid protocol and application development, as well as greater synergy between various research groups. Run-time sharing refers to the sharing of code and resources such as memory and radio, and will allow several protocols to co-exist without burdensome memory requirements or contention problems. Even though most current applications are simple and require just a single network protocol, future developments may result in usage of multiple ones in the same network.

To accomplish these goals, we start by outlining the services provided by and functionalities implemented in the network layer. These requirements lead to a componentized network layer consisting of reusable modules from which we can easily construct network-layer protocols.

It is challenging to find the right granularity at which to break up functionality at the network layer; a very fine-grained decomposition will incur unnecessary runtime overhead, while too coarse a decomposition will not leverage all of the possible sharing and may result in significant reimplementation. In Section 3, we present why and how we modularize routing protocols to architect network layer, and we describe the basic modules in

^{*} Computer Science Division, University of California, Berkeley

[†]Arch Rock Corporation

[‡]International Computer Science Institute (ICSI)

Section 4.

As we show in Section 5, we have successfully implemented multiple published sensornet routing protocols in our architecture. The ability to accommodate a wider variety of existing protocols hopefully minimizes changes with the advent of new ones. We also implemented a number of previously inexistent variations of these protocols by replacing specific modules². Furthermore, the performance cost of this modularity must be low, otherwise designers will circumvent the proposed interfaces, undermining any benefits of the architecture. Thus, in Section 7 we quantify both the performance costs for our modular implementations (which are minimal) and the reductions in protocol-specific code (which are significant). We conclude in Section 8 with a discussion of future challenges for a sensornet architecture.

2 Related Work

This paper builds on two previous pieces of work. The creation of an overall sensornet architecture was proposed in [4]. Following the example of the Internet architecture, where the "narrow waist" allowed rapid innovation both above and below IP, the paper starts by trying to locate the narrow waist of a sensornet architecture. The Internet's narrow waist, IP, provides the abstraction of point-to-point (or point-to-multipoint) best effort packet delivery. This is not a suitable unifying abstraction for sensornets because they have a far wider variety of packet delivery models - such as convergecast (many-to-one), dissemination, data-centric routing, data-centric storage, and others - some of which employ application-specific processing at each hop. Furthermore, IP also provides a standard addressing scheme which is insufficient for sensornets. Given this networklayer diversity, the natural location for the sensornet narrow waist lies lower, between the link and network layers. This idea was substantiated in the SP proposal [21] as a unifying link layer abstraction. In this paper we build upon SP and address the next natural step by proposing an architecture for the network layer in sensornets.

Our work is also inspired by a number of prior efforts in creating modular systems in different contexts. The x-Kernel [10] is an operating system and framework for protocol implementation that combined composability with performance. It focuses on high-performance communication among complete, stand-alone protocols, whereas we attempt to distill many protocols to their common elements in order to maximize reuse.

The Click modular router [15], on the other hand, proposes a flexible composition model for packet-processing modules that enables fine-grained extensions to the forwarding path of an IP router. In contrast, we attempt to modularize entire protocols at a level coarse

enough to reduce the effort required to piece separate components together, yet fine enough to ensure flexibility of combinations. We believe that the components proposed in this paper can be constructed from Click-like elements and are complementary.

Maté [17] is a virtual machine enabling efficient dissemination of code in sensornets. Much like Click, it is concerned with low-level modularity, and does not focus on the definition and construction of multiple network protocols. It is a tool for code dissemination, and as such can be used to distribute network protocols in sensornets. We believe that Maté is complementary to our work.

The work by Condie et.al. in [3] deals with composable transport layer protocols for DHTs, and employs a dataflow model akin to Click. The authors observe that highly-distributed Internet systems exhibit more diverse traffic patterns, and are more suitable for modular protocol designs. We can draw a parallel with the network layer in sensornets, where one of the motivations for modularity is diversity. However, they only deal with the data path, treating routing as a black box. Furthermore, their primary focus is on ease of experimenting with protocol variations, and not run-time code reuse and sharing.

MACEDON [22] provides a framework to describe overlay protocols in the form of finite state machines, enabling the concise expression of any overlay protocol resulting in ease of implementation. P2 [19] allows for declarative specification of new overlay protocols. Both MACEDON and P2 focus on reducing the effort required to generate a single overlay network layer, whereas we also consider co-existing ones.

Finally, Aspect-Oriented Programming (AOP) [13] is a programming technique that allows for isolation, composition and reuse of code that cross-cut different objects or modules. In this paper we focus on network layer properties that can be cleanly separated and encapsulated into distinct components, leaving those that affect multiple components in systemic ways to future work.

3 Modular Network Layer

In this section, we present our modular network layer that aims to achieve the two goals set forth in Section 1: (1) code reuse, and (2) run-time sharing. Achieving these goals fundamentally requires one to decompose the network layer into smaller components that can be re-used by various protocols.

To motivate and provide intuition behind our decomposition, consider the five network protocols shown in Figure 1(a). While these network protocols expose different service interfaces and come with their own implementation, a more careful inspection reveals multiple commonalities among them (Figure 1b). Identifying and encapsulating common functionalities would make

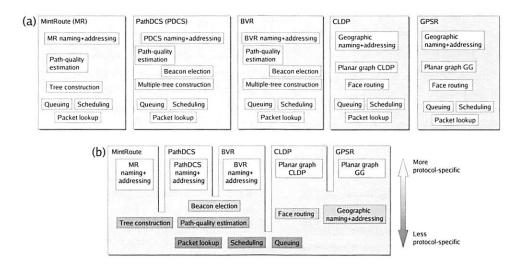


Figure 1: Basic decomposition of five protocols. (a) Current situation: all functions are protocol-specific. (b) Common functions can be shared, reducing the total number of components in the system.

it both easier to build new protocols and enable multiple protocols to share components during run-time.

In the following subsections, we address three questions that are central to defining a network layer:

- What are the services provided by the network layer? (§3.1)
- What are the components of the layer, and what functionality does each component implement? (§3.2)
- How do the components interact with each other, and what is the packet format? (§3.3 and §3.4)

3.1 The Network Layer Service

Similar to IP, the network layer in sensornets provides a best-effort, connectionless multihop communication abstraction to higher layers. However, unlike IP, the network layer in sensornets exposes different addressing and naming schemes, which are required to implement various communication abstractions. Figure 2 summarizes the services provided and functionalities implemented by the network layer.

To provide these services, the network layer needs to implement a variety of functions. These functions can be classified into two categories: control plane and data plane. Control plane functions include identifying and addressing nodes, as well as route discovery and maintenance. Data plane functions include packet forwarding, queue management, and packet scheduling.

In our design we assume that the network layer sits above the sensornet protocol (SP), the narrow waist of the sensornet protocol stack as proposed by Polastre et

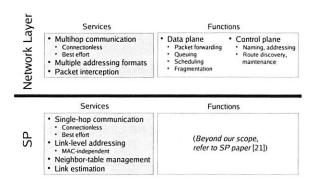


Figure 2: Services provided by, and functionalities implemented in, the network layer, as well as services provided by SP.

al. [21]. The lower part of Figure 2 summarizes the services SP provides to the network layer.

3.2 Network Layer Components

A key question in defining the network layer decomposition is the granularity we should achieve. We strove for one coarse enough so that closely related functions were grouped together (such as topology creation and maintenance), while still providing the flexibility to maximize code-reuse when implementing protocols. We began with a coarse-grained decomposition, and progressively split these components up in order to reach this desired granularity. Further, we attempted to create narrow and well-defined interfaces in order to avoid dependencies, allow for interchangeability of components and minimize composability constraints for new protocols.

At the first level, we follow the natural decomposition of the network layer into separate control and data plane components. Not surprisingly, this is similar to the way

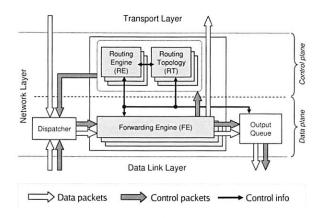


Figure 3: The network layer decomposition, with the flow of packets and control information among the components.

the software is structured in today's IP routers. However, this is too coarse-grained, as it enables little code reuse.

Of the two components, the control plane is typically far more complex than the data plane, as it needs to implement non-trivial functionalities such as topology discovery and routing. To facilitate further reuse, we split the control plane into two components: *routing engine* (RE), and *routing toplogy* (RT). RT is responsible for discovering and maintaining the network topologies, examples of which include trees, multi-trees, and meshes. Once the topology is created, RE computes and maintains routes over the topology. This decomposition allows one to reuse various routing protocols with different topologies. For instance, the data-centric routing protocol PathDCS [6] and the point-to-point protocol Beacon Vector Routing (BVR) [8] both use a multi-tree topology, which can therefore be reused.

While the data plane is in general simple (compared to the control plane), our second goal of achieving run-time sharing induces a natural decomposition of the data plane as well. Upon the arrival of a packet, the data plane needs to obtain the next hop(s) to forward the packet from the control plane (i.e., RE in our case). If multiple packets need to be forwarded at the same time, the packets have to be enqueued and scheduled appropriately. This suggests a decomposition of the data plane into two components: *forwarding engine* (FE) that obtains the next hop(s), and *output queue* (OQ), which implements buffer management and packet scheduling across different protocols.

Figure 3 shows our decomposition and the the interaction between components. We discuss the services and functionalities implemented by each of these components below, and provide examples in the next section³.

Output Queue (OQ) The OQ module performs buffer management and packet scheduling across all packets

forwarded by the node. Different queuing disciplines, as well as network-level transmission scheduling, can be implemented in the OQ. For co-existing protocols to use the communication resource fairly (as defined by the implemented policy), only one OQ module can be in use at any one time. This is in contrast to the FE, RE, and RT modules, multiple instances of which can operate simultaneously on the same node.

Forwarding Engine (FE) The main function of the FE is to obtain the next hop to which the packet is to be forwarded. In the case of multicast communication, multiple next hops will need to be obtained. This is achieved by having the FE query the corresponding RE based on the protocol used. Subsequently, the packet is sent to the OQ to be forwarded. The FE is agnostic to naming and addressing, to maximize module replaceability.

Other functions of an FE include local delivery when the RE determines that the local node is the destination, hooks for interception of packets for purposes of innetwork aggregation, network level retransmission, and multicast. Finally, the FE may opt to perform buffer management and packet scheduling across packets belonging to the same network protocol. In contrast, the OQ operates on all packets traversing that node.

Routing Engine (RE) The RE provides naming and addressing services to the higher layers, and is the only component in the system that understands the protocol's address format. Functions implemented in the RE include (1) determining whether a packet should be forwarded, has reached its destination and thus be accepted, or dropped, (2) if the packet is to be forwarded, the next hop(s) given its destination. The RE implements the logic for determining routes given a destination, using information about an abstract representation of the network topology given by an RT module.

Routing Topology (RT) RT modules are responsible for creating and maintaining basic communication abstractions, with related routing information used by REs either to determine next hops or to construct more complex protocols. The RT is the module that will exchange control traffic with RTs in other nodes, for determining and maintaining the network topology. Examples of communication topologies are trees, geographic coordinates, or any node labeling allowing routes to be found.

We make the distinction between the RE and RT clearer with an example. One common communication topology is a tree, used for sending data to a basestation. There are several ways to build a tree: they can vary in characteristics with respect to stability, convergence, balance, or whether the tree is periodically maintained or is a one time construction. These would correspond to different RTs providing the same abstraction. An RE, on the other hand, performs the actual lookup process to de-



Figure 4: Network packet header format.

termine the next hop(s) to a destination, and is coupled to a topology class and not a particular RT.

3.3 Interfaces

We now provide high level descriptions of the network layer service interface as well as interfaces between the four major components.

Modules in the data path, namely FE and OQ, pass complete packets around and thus their corresponding interfaces are narrow. On the other hand, the FE/RE interface consists of two basic calls: one to obtain the next hop(s) for the packet given its destination, and the other to obtain the cost-to-destination from the current node⁴.

The RE interacts with the RT to obtain the necessary information for determining routes. In defining the division between the RE and RT, the diversity in routing algorithms and communication abstractions in sensornets becomes apparent. Creating a unified interface between the two components would have increased code size, added complexity, and enabled untested and potentially unstable combinations of RTs and REs. Consequently, this interface will be somewhat more protocol specific, in the sense that each class of communication topologies will export a standard interface. On the other hand, a programmer can independently decide to split the RT component by adding a shim layer that would export a standard interface to the RE if he wants.

Finally, the components layered above interact with the network layer by specifying a protocol, an address, and providing a data packet to be sent. This service interface exposes the protocol-specific network address, which is subsequently interpreted by the RE.

3.4 Packet Header Format

Network protocols rely on node coordination to implement their services. This coordination requires interaction between similar components at different nodes, and is typically enabled using information carried in the packet headers. Thus, the packet header format ultimately dictates how components at different nodes interact with each other.

The main issue we are faced with when designing the format of the packet header is the portion of the header each component can access. Our decision is to associate a sub-header with each component involved in forwarding the packet, and allow a component to access only its own header with the rest being opaque. This way, we avoid unnecessary bindings, and make it easier to interchange different components. For example, the destina-

tion address is only understood by RE. This allows the same routing engine (for example, one that routes based on geographic coordinates), to be combined with different forwarding engines, such as opportunistic [1]. Except for the network protocol identifier, sub-headers may be absent depending on the corresponding components.

Figure 4 shows our packet header format. Since multiple network protocols can exist in a sensornet, we use a *protocol identifier* to select the appropriate components. This identifier is the only required field in the packet header. The rest of the header consists of three subheaders, one for each component (OQ, FE and RE) involved in packet forwarding. The size and the format of each of these headers is component-specific, and we give a brief description of each below.

- 1. The OQ header contains scheduling and buffer management information (*e.g.*, packet priority) and is interpreted by the OQ module.
- 2. The FE header contains information required to forward the packet (*e.g.*, hopcount or a unique message identifier for suppressing packet duplicates).
- 3. The RE header holds information required to determine the next hop.

4 Module Examples

In order to demonstrate the feasibility of the modular network layer, we describe examples of individual components in this section, and show in the next section how they can be composed to implement several of currently available network protocols.

4.1 Output Queue Modules

An Output Queue (OQ) module is the one place in the system which all outgoing packets must traverse, a necessary condition to implement packet scheduling for all network protocols. Thus, while multiple types of OQs exist, only one can be in use in a node at any time. A basic module may implement simple priority scheduling, whereby control packets are given higher priority when queue drops occur and when selecting the next packet to send. More complex scheduling can be implemented to improve end-to-end fairness, as well as to reduce physical-layer contention⁵ amongst neighboring nodes. Finally, scheduling of packet transmission can be influenced by the routing topology, useful in the case of in-network aggregation. We begin with the description of a basic, simple priority scheduler.

The *Basic* OQ provides simple priority scheduling and queue management functionality. Given a packet of high priority, the basic module transmits it before one of

lower priority. The determination of a packet's priority is dependent on the network protocol. For instance, some protocols may require their control packets to be sent as soon as possible and preferably not dropped when the queue is full, and may set these as high priority.

Flexible Power Scheduling (FPS) [9] is a networklevel, time-division multiple access (TDMA) algorithm that aims to provide high utilization and fairness on a per-destination basis. FPS divides time into cycles, with a fixed number of slots in each cycle. In each cycle, FPS allocates slots on a per-destination basis. At the next-level, slots allocated for a particular destination D at node N are divided among the neighbors that forward packets to D through N. This allocation is based on the neighbors' queue occupancies: FPS allocates more slots to the neighbor with more packets destined to it. This policy aims to balance supply and demand at each neighbor, and at the same time achieve high utilization. Note that since FPS requires knowledge of the flow to which a packet belongs, interaction with the RE is necessary: classification of the packet is based on the destination address, the format of which is known only to the RE.

Epoch-based Proportional Selection (EPS) [5] is another example of an OQ. Unlike FPS which uses a static total number of slots per cycle, EPS dynamically adjusts this based on the current demand. EPS enforces fairness using non-work-conserving, weighted roundrobin servicing of children's queues, using the number of upstream nodes of each child as the weights. This number is carried within each data packet transmitted in the OQ header. Similar to FPS, EPS requires classification of packets based on their network destination, thus interaction with the RE is necessary as well.

Finally, unlike the FPS and EPS components, the *Epoch* module's transmission schedule is determined by external components such as the routing topology. Using this knowledge, the Epoch module allows nodes further away from the destination to transmit before the rest, enabling aggregation of data at each intermediate hop towards the collection node. This reduces the total number of packets transmitted and thus energy consumed.

4.2 Forwarding Engines

Forwarding Engines (FEs) are components that are more protocol-specific in the data-plane, determining how and when packets are to be forwarded. Opportunistic forwarding and multicast are examples of FEs. We begin by describing a basic forwarding engine.

The *Basic* FE obtains per-packet next-hop information from the corresponding RE and checks for packet interception requests from higher layers. Additional functions include detection of routing loops and suppression of duplicate packets.

The *Opportunistic Forwarding* engine implements perpacket suppression functionality similar to ExOR's [1]. Packets eligible for forwarding include those received from neighbors further away from the destination. These packets are held onto for a pre-determined period of time; if no similar packets from nodes equally far or closer to the destination are received within this time, the packet is sent, otherwise its transmission is suppressed.

Note that the precise next-hop neighbor need not be identified. Instead, we require knowledge of the cost to destination, which can be provided by certain REs.

Multicast FEs forward multiple copies of the same packet to different next-hop nodes. These FEs only provide the functionality, they do not decide whether a packet should be multicast. This decision is made by the RE during packet lookups, when the RE implicitly indicates that multicast should take place by returning the list of all next-hops.

4.3 Routing Engines

A Routing Engine (RE) can build upon basic communication abstractions provided by Routing Topologies (RTs) to construct more complex ones. MintRoute, PathDCS and Beacon Vector Routing (BVR) are examples of such REs. On the other hand, simpler ones such as Broadcast can operate independent of RTs.

The *Broadcast* RE handles all packets that are logically broadcast to all one-hop neighbors. Thus, this simple RE does not provide any specific next-hop to which a packet should be forwarded, neither does it provide a cost-to-destination metric. It is a basic RE that can be used by all protocols, either at the transport or network layer, that require logical one-hop broadcasts.

The rest of the protocols, PathDCS, MintRoute and BVR, will be discussed in greater detail in Section 5. Briefly, PathDCS [6] provides data-centric routing capabilities using routing trees rooted at random nodes (beacons). Each piece of data is mapped onto a network path using the beacons as guides, and the destination node for that data is the terminating node of the path. BVR [8] on the other hand uses the same many-to-one routing abstraction to construct a logical coordinate system based on hop distances from the roots of these beacons. For simpler protocols such as MintRoute [24], the RE can be very light-weight since there is little additional function-

ality to be implemented. We note that in general REs are more specific to the network protocol than FEs or RTs.

4.4 Routing Topology

Communication abstractions can be composed from a few basic Routing Topologies (RTs), which REs can use to construct more complex ones. Interfaces provided by RTs can vary significantly with the abstractions provided. Examples of RTs include MTree, Gradient and Geographic, which we describe below.

The MTree RT provides many-to-one routing abstraction using trees. The primary metric used in tree construction is the minimization of expected packet transmissions to the root. Periodic control information exchanged between neighbors determine the bidirectional, one-hop as well as end-to-end path quality. MTree constructs M routing trees rooted at random nodes in the network, except for the first which is at the base, or collection, station. Thus, this module can be also used for basic route-to-base applications.

Since MTree maintains routing information in the form of routing tables, it can provide hop-distances to each root. This information can be used by BVR to construct a logical coordinate system on which point-to-point routing can be implemented, by a basic FE to detect loops, or by PathDCS to determine the maximum number of hops to take towards a certain beacon (root). Next-hop information can be used by FPS to determine the parent from which supply slots can be requested.

The *Gradient* topology is similar to MTree in that each node maintains its cost-to-destination. However, this module does not specifically determine the next-hop to which a packet should be forwarded. Such a topology is simple to construct and maintain, and is useful for purposes of scheduling and opportunistic forwarding. The Epoch module, focusing on scheduling, is advantageous for in-network aggregation of data as mentioned earlier. Opportunistic Forwarding, which requires just the cost-to-destination and not the specific next-hop, can use the function provided by Gradient as well.

Finally, the *Geographic* RT provides geographic coordinates via, for example, the Global Positioning System. This module can be used by multiple other components: by a GPSR-like RE to provide point-to-point routing, or by the application to determine, say, the location of a fire. The Geographic RT can be augmented with more functions. For instance, it can periodically probe neighboring nodes to obtain their coordinates, enabling it to provide information such as the closest next-hop node towards a given destination.

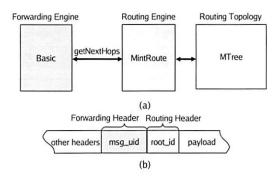


Figure 5: (a) Main modules in the implementation of MintRoute. (b) Packet header contents.

Also, Euclidean distance to destination can be provided as the cost metric.

5 Composition of Protocols

We now describe how various existing protocols can be composed from the modules providing services and implementing functions required of the network layer. Table 1 provides a summary of several protocols and their corresponding components. Since there are modules that, by their very nature, cannot interoperate efficiently or at all due to their inability to provide required functions, there exist constraints that restrict possible module combinations. We elaborate on this at the end of this section.

5.1 Collection

MintRoute [24] is a collection protocol that routes packets towards the root of a tree, and is the basis for many data gathering applications. A message is recursively forwarded to the current node's parent until the root is reached. Figure 5 shows the main components that make up MintRoute. We use Basic FE that provides routing loop detection ability, MTree as the RT, and a MintRoute-specific RE.

Although multiple routing trees are provided by MTree, MintRoute uses that rooted at the collection point. Similar to the monolithic version of MintRoute, the destination address is therefore implicit and need not be included in the RE header. To remove duplicate packets due to retransmissions, the FE uses unique identifiers placed in the FE header. Finally, the OQ provides simple priority scheduling, and can give higher priority for control traffic from MTree than for data traffic.

A variant of collection protocols involves in-network aggregation of data. Synopsis Diffusion (SD) [20] is one such protocol, providing duplicate-insensitive aggregation in sensor networks. A gradient is set up originating from the destination node, with nodes further away sending packets earlier so that those closer can aggregate before they forward. This reduces the total number

Table 1: Decomposition of current and composition of new network protocols. Implemented mod-

ules in italics.

Network Protocol	Output Queue	Forwarding	Routing	Topology
		Existing P.	rotocols	
MintRoute	FPS/Epoch/Basic	Basic	±.	MTree
GPSR	Basic	Basic	GPSR	Geographic Coords +
				GG/RNG Planarization
CLDP	Basic	Basic	GPSR	Geographic Coords +
				CLDP Planarization
PathDCS	Basic	Basic	PathDCS	MTree
BVR	Basic	Basic/Re-xmit6	BVR RE	MTree
Synopsis Diffusion ⁷	Epoch	Basic	-	Gradient
Directed Diffusion	Basic	Multicast	Directed Diffusion	
AODV	Basic	Basic	On-demand point-to-point	-
GRAd	Basic	Opportunistic	On-demand point-to-point	-
ExOR	Basic	Opportunistic ⁸	OSPF	=
Trickle ⁹	Basic	Basic	Broadcast	•
		New, Hybrid	Protocols	
Opp. MintRoute	FPS/Epoch/Basic	Opportunistic		MTree/Gradient
Alternate Paths	Epoch/Basic	Basic/Re-xmit	-	MTree
Scoped Trickle	Basic	Basic	Scoped Broadcast	<u>□</u> 1

of transmissions by each node. The natural RT to use is thus Gradient, and the scheduling mechanism can be provided by the Epoch OQ with information from Gradient. At intermediate hops, data packets have to be intercepted by the FE and sent up the stack for aggregation. Thereafter, the packet is scheduled for transmission in the OQ module.

5.2 Point-to-Point

We next describe two classes of point-to-point network protocols in sensornets, based on either logical or actual geographic coordinates. We begin with the latter. There is a large number of variations on Geographic Routing [2, 12, 16, 14], and we present the basic idea here. Each node maintains knowledge of its coordinates as well as those of its neighbors'. Next-hop(s) to which a packet is forwarded is(are) determined using the destination's coordinates carried within the RE header.

Two routing phases exist, greedy and face routing. In greedy routing, nodes forward the packet to the neighbor closest to the destination. If the current node is closest compared to all its neighbors, the forwarding node switches to the next phase: face routing. Packets are then forwarded along the face edges of an underlying planar graph, changing faces when appropriate and applying rules that guarantee progress towards the destination. In this phase, additional state has to be carried in the packet 10 : the current phase, the node's coordinates when face routing begun (L_p) , as well as the coordinates (L_f) and the edge (e_0) where the packet entered the current face. At each step the node checks if greedy can resume and does so if possible.

Figure 6 shows the modules implementing geographic

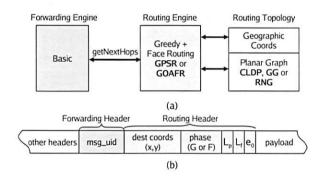


Figure 6: (a) Main modules in the implementation of Geographic Forwarding. Notice that there can be many variations (GPSR, CLDP, GOAFR, GOAFR+CLDP), with reuse of many modules. (b) Packet header contents. The routing header includes state for face routing.

routing and the contents of their corresponding headers. The routing topology components provide two abstractions: (1) Geographic Coordinates, which maintains and provides the coordinates of the current node and its neighbors, and (2) Planar Graph, which provides a planarized version of the underlying connectivity graph. The Planar Graph functionality can, for instance, be provided by CLDP, Gabriel Graph, or Relative Neighborhood Graph. The RE is responsible for determining the next-hop to the destination, and maintains state in the RE header. By replacing the face routing rule, the RE can implement GPSR or GOAFR routing, reusing the RT.

In contrast with Geographic Routing, Beacon Vector Routing [8] (BVR) uses a topology-based logical coordinate system to find routes. A subset of the nodes is chosen as *beacons*, and all nodes in the network learn their distances in hops to each of these beacons. Routing is performed in a greedy fashion by minimizing a

distance function on the coordinates. Next-hop information can be determined using the destination's, current node's and neighbors' coordinates. BVR's RE consists of a simple module that derives coordinates from MTree and computes the distance function between two coordinates. Routing is performed in three phases. First, greedy routing is attempted. If that fails, packets are routed towards the root closest to the destination. Upon reaching that root, scoping flooding is performed. The routing header consists of the destination's coordinates and identifier, the minimum cost the packet has seen so far, the current routing phase, and the time-to-live for scoped flooding. The FE used by the original BVR tries to send messages to alternate next-hops upon transmission failure. The BVR RE can also be paired with other forwarding disciplines, such as opportunistic, since the cost-to-destination can be obtained from the virtual coordinates created

5.3 Data-Centric

PathDCS [6] is an example of a data-centric routing protocol. Rather than map data onto specific locations in a geographic or logical coordinate system, PathDCS maps it onto a path through the network. This path is divided into parts, or segments, which is in turn defined by a certain number of hops to take towards a particular network beacon. The storage location for the data is then the terminating point of the concatenated segments. Mapping data storage location onto terminating nodes of existing paths ensures that a node always exists at that location. This eliminates the need to know the network boundaries, a requirement necessary for coordinate-based systems.

The PathDCS network protocol can be decomposed into a protocol-specific RE that runs atop the MTree topology. Its RE header contains the data key, the current segment being traversed, and the current number of hops to traverse towards the next beacon. PathDCS can use other kinds of FEs, such as Opportunistic, and is compatible with OQ modules providing scheduling functionalities.

Directed Diffusion (DD) [11] is another example of a data-centric routing protocol. DD names data using attribute-value pairs, and interests for data are disseminated through the network. The dissemination process sets up gradients allowing nodes with the relevant data to forward them to the querying nodes. In Directed Diffusion, multiple overlapping interests can be aggregated, enabling data to be sent once from the source to the aggregation point before being duplicated for forwarding towards each interest source. Thus, DD is primarily composed of an RE that sets up gradients from each interest source, as well as a multicast FE. In this case, no routing topology module is required.

5.4 Dissemination

Trickle [18] is a code dissemination and maintenance protocol. Since the objective of Trickle is to have all nodes in the network run the same code, it can be considered a transport layer protocol that implements one-to-all reliable transfer of data. To support Trickle, the network layer provides a simple one-hop broadcast RE. A received, logically broadcast packet is passed to the Trickle transport layer. Trickle subsequently uses delayed transmission and suppression to control the rate at which messages are broadcast in the network. Although functionally similar to opportunistic forwarding, Trickle's suppression mechanism requires transport layer knowledge, and therefore is not placed in the FE.

5.5 New, Hybrid Protocols

The network layer modularity simplifies the creation of new protocols by swapping one component for another, or making slight modifications (Table 1). For instance, in the case of the collection protocol MintRoute, we can route to any of the roots provided by MTree by including the destination root address in RE's header (Figure 5b).

Network-level retransmissions can also be added by having the RE return all next-hops closer to the destination. An FE providing the retransmission function can attempt to resend packets to alternate next-hops. Such an RE can also be used by an FE that probabilistically selects a next hop to balance forwarding load. Lastly, replacement of the basic FE with the Opportunistic FE yields opportunistic collection routing.

A different kind of collection protocol can be implemented using the Gradient RT and the Opportunistic FE. The former readily provides knowledge of the current node's distance to destination, required by the latter. One can imagine various performance benefits to be gained from such a pairing, but the evaluation is beyond the scope of this paper.

5.6 Composability Constraints

We end this section by looking at the constraints encountered when attempting to combine different routing, forwarding, and output queue modules. In the following discussion we use the term routing to mean the combination of RE and RT. Table 2 lists the combinations that are feasible and those that aren't. We see that the basic OQ works with all protocols, as does the basic FE. Network-level retransmission is a functionality embedded in the FE, but is effective only if routing provides multiple distinct next-hops. From column *Re-Xmit*, this is not the case for Table Driven and Broadcast routing. Opportunistic forwarding, in turn, requires a globally meaningful cost-to-destination metric. PathDCS, MintRoute, Gradient, Geographic, and BVR routing pro-

Table 2: Constraints on the composition of routing engine/topologies with forwarding engines and output queuing. The composition may be $good(\sqrt{})$, not optimal (*), or not possible (-).

	Forwarding Engine					Output Queuing		
Routing	Basic ¹¹	Re-xmit ¹²	Opport.13	Multicast	Basic ¹⁴	Epoch	FPS	
MintRoute	\checkmark	\checkmark	\checkmark	-			\checkmark	
Gradient	\checkmark	\checkmark	\checkmark	-	\checkmark	\checkmark	\checkmark	
Table Driven	\checkmark	72	-	_	\checkmark	*	*	
PathDCS	\checkmark	\checkmark	*	-	\checkmark	\checkmark	\checkmark	
BVR	\checkmark	\checkmark	$\sqrt{15}$	-	\checkmark	*	*	
Geographic	\checkmark	\checkmark	\checkmark	-	\checkmark	*	*	
Broadcast	\checkmark	2	-	-	\checkmark	*	*	
Directed Diffusion	· · ·	_	-	\checkmark	\checkmark	*	*	

vide such a cost field for each destination, although for the last two there are local minima, and routing solely based on these may not lead to the destination. Finally, while both Epoch and FPS are currently designed for collection routing, and thus will work optimally only with MintRoute and Gradient routing, they can be extended for the case of multiple destinations.

6 Completing The Picture

The previous sections discussed details of the major parts of the network layer, but at the high level. In this section we discuss the rest of the components and bring everything together by providing a short description of the actual packet forwarding process.

6.1 Miscellaneous Components

Four additional components, the Protocol Service, Network Service Manager, Buffer Manager, and the Dispatcher, are minor components but are still essential to the network layer. We describe them below.

Protocol Service As described in Section 3, we decomposed each network protocol into a Forwarding Engine, Routing Engine, and possibly a Routing Topology. To simplify the usage of these components, we wrap them in a Protocol Service module, an instance of which exists for each different network protocol. This module provides a unified service interface to higher-layer programmers and specifies the necessary connections between the wrapped components.

Network Service Manager The Network Service Manager aggregates and maintains information related to service requests originating from higher-layer components. For instance, applications can register with the service manager hooks to intercept certain protocols' packets. The service manager is then queried by the forwarding engines to determine if interception is necessary. Since there may be multiple applications requiring such

services, and since these functionalities are required of all protocols, we gather them into a single module.

Buffer Manager The traditional way of managing buffers, that is, RAM space required to hold packets received or due for transmission, is to have each component statically allocate space at compile time. Sharing of buffers between different components is not possible, thus in general the system is less able to accommodate bursts of buffer requests and usage of available buffers are less efficient. The Buffer Manager tackles this issue by aggregating available buffers from all components at node initialization time, and provides them on request based on some user-defined policies. Since this module works across layers, it is not part of the network layer, but instead simply provides buffer aggregation services. Thus, in-depth design and evaluation of this module is beyond the scope of this paper.

Dispatcher The dispatching module maintains a protocol table with an entry for each network protocol running on the node. Upon receiving a packet, the packet's protocol identifier is used to determine the corresponding forwarding engine which the packet should be sent. This protocol table will generally be configured at compile time and be uniform across a sensor network.

6.2 Packet Forwarding Procedure

To show how various parts of the system fit, we next describe the forwarding process undertaken upon reception of a packet using Figure 7.

- Messages first arrive at the dispatcher either locally or from the network. The dispatcher determines the protocol identifier, either from the higher-layer component if local, or otherwise from the message itself.
- The message is subsequently sent to the corresponding FE based on the identifier.

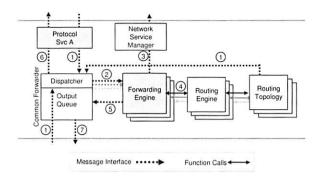


Figure 7: Data path in the network layer.

- The FE checks whether an application has registered to intercept messages of this protocol, if so the message is handed to the application, otherwise
- 4. the RE is queried to determine the specific next hop(s) for the message, or provide some cost value to determine if the current node is closer to the message destination. Since the RE is the only component that understands the address format, it can attach the identifier of the flow to which the message belongs for purposes of scheduling later in the OQ.
- FE then sends the message to the OQ specifying whether it should be forwarded or sent to a higherlayer component.
- At the OQ, if the message is determined by the RE to be destined for this node, or by the Network Service Manager to be intercepted, it is sent up the stack, otherwise
- 7. the message is to be forwarded, and is scheduled for transmission based on the implemented policy.

7 Evaluation

In this section we evaluate a subset of the protocols implemented with the proposed network layer. We revisit our goals of code reuse and run-time sharing of components by multiple protocols, with a minimal penality at storage and performance, and show that they are largely achieved.

7.1 Methodology

Our two basic metrics in the evaluation are code size and memory footprint on the one hand, and forwarding delay on the other. We require only the addition of one byte for the protocol identifier and not the transmission of additional packets. This, coupled with the fact that energy consumption is dominated by communication rather than computation in today's sensornets, lead us to believe that this consumption should not increase significantly and thus we do not include it as a metric.

To quantify real benefits and costs, we compare the implementations of just three protocols (due to space constraints), PathDCS, BVR, MintRoute. These implementations include the original, monolithic version (Original), implementation over SP without the network layer modularization (SP),²⁰ and our modularized version running on the network layer, on top of SP (NLA & SP). All evaluations are performed on TelosB motes [25].

To measure the forwarding cost, defined to be the delay encountered by a packet when it first enters the network layer until it is sent to the link layer for transmission, we instrumented the code to read a hardware clock with the granularity of microseconds, properly accounting for the (small) measurement overhead.

7.2 Code size and memory footprint

One of the main objectives of creating a modular network layer for sensor networks is to increase code reuse, thereby allowing for multiple protocols to coexist cleanly and efficiently on a single node and simplifying future network protocol design. There is some overhead, however, to allow demultiplexing and configuration among different modules, when compared to the monolithic implementations. Table 3 compares code and memory sizes between the different implementations. Code size refers to the amount of program memory occupied by the networking code, while memory footprint refers to the amount of RAM allocated. The former is important since it is desirable that non-application code not consume significant amounts of limited memory. Volatile memory, however, is more crucial, since there is a direct relationship between RAM and energy consumption.

Three observations can be made from Table 3:

- 1. When we combine different protocols we observe clear gains: the two combinations use up 40% and 58% less memory, and occupy 18% and 37% less program memory. Of course, the gains are limited by the extent to which protocols share underlying primitives, but we note that our network layer realizes these gains, and that we have seen considerable potential sharing among existing and new protocols (c.f. Table 1).
- For the individual protocols (upper section), code size and memory size are comparable. As we will show next, much of the difference in code size resulted from additional functionalities. The memory consumption numbers are similar, which is good from an energy-conservation perspective.
- The last observation relates to code reuse at development time. We have decomposed network protocols such that protocol-specific code is limited to a

Table 3: Code and Memory Size Comparison of Architectures

Network Protocol ¹⁶	Code Size ¹⁷			Memory Footprint ¹⁸		
	Mono	SP	NLA & SP	Mono	SP	NLA & SP
MintRoute	2562	2356	6140(92)	1400	1273	1862
PathDCS	6786	4968	6450(988)	1764	1981	1766
BVR	642219	-	9060(1512)	1411	-	1889
Synopsis Diffusion		-	3686	200	-	872
	Prot	ocol Coe	existence			
MintRoute + PathDCS	9348	-	7684	3164	-	1894
MintRoute + PathDCS + BVR	16430	-	10354	4575	-	1917

single component in most cases, typically the routing engine. The numbers in parenthesis in Table 3 show the code sizes that are protocol-specific and not likely to be reusable. Protocol-specific code is substantially less in our implementation²¹. For the Original and SP versions, these numbers are essentially the same: most of the code is not reusable as a result of tight integration, or unusability of modules by other designers due to the lack of common interfaces or unclear division of functionalities implemented in modules. In the case of the SP implementations, neighbor-table management, and link estimation are moved into SP itself, but the remaining network layer code is again protocol-specific, hindering substantial code reuse.

We now return to the increase in code size observed for BVR and MintRoute, by examining, with reference to Table 4, how the Original and NLA implementations of BVR are decomposed. The decomposition of MintRoute presents similar trends and is omitted for brevity. The modules are grouped by approximate functionality. It is comforting that the code implementing primary BVR functions (Core Protocol Code) are similar in size. The communication stack on which each implementation sits has a queue and a link estimator in the original version. In addition, SP provides neighbor-table management, simple one-hop scheduling, and duty cycling²² of the radio. These additional features account for SP's larger code size. The last group, Additional Features, are unique to our implementation, and account for most of the difference in total code size. These include dynamic demultiplexing, tools for maintaining header independence, multiple queues for buffers²³, and dynamic memory management. We note that not all of these extra features are a requirement of the network layer, and that in most cases one can implement simpler and smaller modules.

Finally, we look at each individual component, drawn from a general library which developers can use to assist in creating new protocols. Table 5 provides the code size and memory footprint for these components. In general, we consider routing engines to be the heart of network protocols, and are thus less likely to be reused. On the

Table 4: Detailed Comparison of BVR Implementations

Monolithic		NLA & SP		
Component	Size(B)	Component	Size(B)	
	Core Prot	ocol Code		
Router	1194	Routing Engine	1512	
Topology State	2638	Routing Topology ²⁴	2136	
Coordinate Table	1422	Coordinate Table	1422	
Coordinate Functions	754	Coordinate Functions	754	
		Forwarding Engine	504	
	6008		6328	
Unde	rlying Com	munication Stack		
Queuing Buffer	394	SP	4244	
Link Estimator	2856			
	3250		4244	
A	dditional F	unctionalities		
		Service Manager	490	
		Output Queue	1822	
		NetService	324	
		BufferManagerM	252	
			2888	

other hand, components such as the MTree routing topology are much more general and can consequently be utilized by a wider variety of protocols. These generic components are the key factor in enabling clean co-existence of multiple protocols with a substantially smaller overall code size and memory footprint relative to their monolithic counterparts.

7.3 Performance

Next we evaluate the modular network layer from a performance point of view. Table 6 provides comparisons of forwarding delay, per module, for the original and NLA & SP implementations of MintRoute, PathDCS and BVR.

As one would expect, packets traversing our network layer experience more delay than they do in the monolithic architecture. The additional delays are incurred due to component and layering abstractions: in the monolithic architecture, it is assumed that only one link layer exists at each node. It is thus possible to have the packet header format known to all components in the system. On the other hand, to improve portability and reuse, the network layer uses the *payload* and *length* meta-data fields to indicate the location of the next payload for the next component receiving the packet. This reduces the

Table 5: Code Size and Memory Footprint of Individual

Components

Component	Code Size (B)	Memory (B)
Output Queues		
Basic	1822	396
Epoch	1892	396
Flexible Power Sched.	2696	564
Forwarding Engines		
Basic	384	64
Opportunistic	1830	169
Retransmit	504	64
Routing Engines		
Broadcast	42	0
MintRoute	92	0
PathDCS	988	2
BVR	1512	0
Routing Topologies		
MTree	2766	88
Gradient	372	82
BufferManager	252	84
Network Protocol Service	324	24
Service Manager	490	8

Table 6: Forwarding Cost Comparison

	Mono	NLA & SF)
Protocol	Time(μs)	Component	Time(µs)
MintRoute	65	Routing Engine:	19
		Routing Topology:	24
		NetService:	12
		Output Queue:	573
		Forwarding Engine:	178
			806
PathDCS	181	Routing Engine:	165
		Routing Topology:	24
		NetService:	12
		Output Queue:	573
		Forwarding Engine:	178
		W20 8850 ()	952
BVR	3752	Routing Engine:	366
		Routing Topology:	2795
		NetService:	12
		Output Queue:	573
		Forwarding Engine:	178
			3924

need to know, say, every possible MAC header in existence. The tradeoff is the necessity of additional operations required to access these fields. Furthermore, additional overhead due to function calls is incurred since the network layer is composed of multiple small modules instead of just one that is tightly integrated.

We observe a fixed overhead cost induced by NetService, Output Queue, and Forwarding Engine components that is significant compared to the cost of the simpler protocols. For BVR, which involves more complex lookup processing, the relative overhead is considerably smaller, representing a 4.3% increase in forwarding time. These numbers must be placed in perspective: although performance is important, unlike the Internet, it is not the primary goal. Most applications we see are very *low data*

rate, low duty-cycle, and the cost in terms of energy of processing a byte is low compared to the cost of sending a byte over the radio. On the same platform, it takes at least 6.25ms to forward a common packet of about 40 bytes²⁵, and thus even with BVR we are still operating under "line speed". In none of these examples will packet processing be a bottleneck. Lastly, the components can be further optimized to reduce processing time.

8 Conclusion

In this paper we proposed, implemented, and evaluated a modular network layer for sensornets that aims at maximizing composability and reusability of protocol modules. We verified that many existing protocols fit naturally in the architecture, and that less effort is required to create new ones. Through evaluation of modules and protocols implemented in the new network layer, we are able to obtain up to 58% memory reduction and 37% less code when running protocols concurrently. Furthermore, we believe that the additional processing latency incurred is acceptable in the context of sensornets.

Looking forward, the work towards a sensornet architecture is still incomplete. One consequence of establishing more strict layering is that certain functionalities, like power management, security, reliability, and time synchronization, need to be accessible to multiple layers. Our hope is to address these cross-layer issues in the future.

Notes

- ¹As evidence of that, the original implementations of four routing algorithms MintRoute, PathDCS, BVR, and CLDP have four different and incompatible implementations of common modules such as link estimation, neighborhood and queue management.
- Note that our goal is to facilitate the implementation of networklayer solutions, it does not actually, say, make end-to-end transfer of data more reliable. As such, we do not focus on implementing new functionalities.
 - ³Minor components, like the dispatcher, are discussed later in § 6.
- ⁴This can be used by protocol implementing some form of opportunistic forwarding.
 - ⁵in the case of shared mediums
 - ⁶Network retransmission to alternate next-hops.
 - ⁷Higher layer computes synopsis.
 - ⁸The complete ExOR algorithm is not implemented.
- ⁹This is just the network layer support for Trickle. The main algorithm runs in the transport layer.
 - ¹⁰See [12] for details.
 - 11 Includes variations, like duplicate detection and max TTL.
 - ¹²Requires alternate next-hops.
 - ¹³Requires global cost-to-destination function.
 - 14 Includes variations like priority, round-robin, fair queuing
 - ¹⁵Does not guarantee delivery due to local minima.
- ¹⁶A dash '-' indicates that no implementation existed in that particular format.
- ¹⁷In bytes. This code size includes only network layer code, thus the code size of SP, approximately 4200 bytes, is not included in the figures for the SP and NLA & SP implementations.

¹⁸In bytes. In order to provide a fair comparison, the memory footprint figures for *SP* and *NLA & SP* implementations include the neighbor table overhead portion of the SP memory footprint, as monolithic implementations maintain their own neighbor table.

¹⁹This figure does not include the code for link estimation, as that feature is not currently provided in our implementation. With that capability included, the code size is 9278 bytes

²⁰The 'MintRoute on SP' combination is the only one in which the original SP implementation [21] is used. All other experiments were run using an enhanced version [23], which has several additional features, and thus, larger code size.

²¹We do not have a figure for Synopsis Diffusion because all of the protocol-specific functionality for duplicate-insensitive aggregation is at the application level. The network layer components are all reusable.

²²Duty-cycling refers to the turning off of the radio from time to time to conserve energy, which can be consumed even when listening to the channel

²³Including queues for different message priorities.

²⁴The RT in this comparison is not MTree, but an equivalent RT ported directly from the original BVR code, to make the comparison closer

²⁵The usual size in a sensornet.

References

- BISWAS, S., AND MORRIS, R. Exor: opportunistic multi-hop routing for wireless networks. In SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications (New York, NY, USA, 2005), ACM Press, pp. 133–144.
- [2] BOSE, P., MORING, P., STOJMENOVIC, I., AND URRUTIA, J. Routing with guaranteed delivery in ad-hoc wireless networks. In ACM Wireless Networks (November 2001).
- [3] CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., RHEA, S., AND ROSCOE, T. Finally, a use for componentized transport protocols. In *Proceedings of the Fourth Workshop on Hot Topics* in Networks (November 2005).
- [4] CULLER, D., DUTTA, P., EE, C. T., FONSECA, R., HUI, J., LEVIS, P., POLASTRE, J., SHENKER, S., STOICA, I., TOLLE, G., AND ZHAO, J. Towards a sensor network architecture: Lowering the waistline.
- [5] EE, C. T., AND BAJCSY, R. Congestion control and fairness for many-to-one routing in sensor networks. In SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems (New York, NY, USA, 2004), ACM Press, pp. 148–161.
- [6] EE, C. T., RATNASAMY, S., AND SHENKER, S. Practical datacentric storage. In *Proceedings of the Third USENIX/ACM NSDI* (San Jose, MA, May 2006).
- [7] ESTRIN, D., GOVINDAN, R., HEIDEMANN, J. S., AND KU-MAR, S. Next century challenges: Scalable coordination in sensor networks. In *Mobile Computing and Networking* (1999), pp. 263–270.
- [8] FONSECA, R., RATNASAMY, S., ZHAO, J., EE, C.-T., CULLER, D., SHENKER, S., AND STOICA, I. Beacon-Vector Routing: Scalable Point-to-point Routing in Wireless Sensor Networks. In *Proceedings of the Second USENIX/ACM NSDI* (Boston, MA, May 2005).
- [9] HOHLT, B., DOHERTY, L., AND BREWER, E. Flexible power scheduling for sensor networks. In *IPSN'04: Proceedings of the* third international symposium on Information processing in sensor networks (New York, NY, USA, 2004), ACM Press, pp. 205– 214.
- [10] HUTCHINSON, N. C., AND PETERSON, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Transac*tions on Software Engineering 17, 1 (1991), 64–76.

- [11] INTANAGONWIWAT, C., GOVINDAN, R., AND ESTRIN, D. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In Proceedings of the International Conference on Mobile Computing and Networking (Aug. 2000).
- [12] KARP, B., AND KUNG, H. T. GPSR: greedy perimeter stateless routing for wireless networks. In *International Conference* on *Mobile Computing and Networking (MobiCom 2000)* (Boston, MA, USA, 2000), pp. 243–254.
- [13] KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspectoriented programming. In *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds., vol. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242.
- [14] KIM, Y., GOVINDAN, R., KARP, B., AND SHENKER, S. Geographic routing made practical. In Proceedings of the Second USENIX/ACM Symposium on Networked System Design and Implementation (NSDI 2005) (Boston, MA, May 2005).
- [15] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. ACM Transactions on Computer Systems 18, 3 (August 2000), 263–297.
- [16] KUHN, F., WATTENHOFER, R., ZHANG, Y., AND ZOLLINGER, A. Geometric ad-hoc routing: Of theory and practice. In *Principles of Distributed Computing* (2003).
- [17] LEVIS, P., AND CULLER, D. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the ACM Conference on Ar*chitectural Support for Programming Languages and Operating Systems (ASPLOS X) (Oct. 2002).
- [18] LEVIS, P., PATEL, N., CULLER, D., AND SHENKER, S. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI) (2004).
- [19] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing declarative overlays. In SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles (New York, NY, USA, 2005), ACM Press, pp. 75–90.
- [20] NATH, S., GIBBONS, P. B., SESHAN, S., AND ANDERSON, Z. R. Synopsis diffusion for robust aggregation in sensor networks. In *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (Sensys)* (New York, NY, USA, 2004), ACM Press, pp. 250–262.
- [21] POLASTRE, J., HUI, J., LEVIS, P., ZHAO, J., CULLER, D., SHENKER, S., AND STOICA, I. A unifying link abstraction for wireless sensor networks. In *Proceedings of the Third ACM Con*ference on Embedded Networked Sensory Systems (SenSys) (Nov. 2005).
- [22] RODRIGUEZ, A., KILLIAN, C., BHAT, S., KOSTIC, D., AND VAHDAT, A. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the* first USENIX/ACM Symposium on Networked System Design and Implementation (NSDI 2004) (San Francisco, CA, March 2004), pp. 267–280.
- [23] TAVAKOLI, A., TANEJA, J., DUTTA, P., CULLER, D., SHENKER, S., AND STOICA, I. Evaluation and Enhancement of a Unifying Link Abstraction for Sensornets. Under submission.
- [24] WOO, A., TONG, T., AND CULLER, D. Taming the underlying challenges of reliable multihop routing in sensor networks. In Proceedings of the first international conference on Embedded networked sensor systems (2003), ACM Press, pp. 14–27.
- [25] Crossbow. http://www.crossbow.com.

Making Information Flow Explicit in HiStar

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières Stanford and UCLA

ABSTRACT

HiStar is a new operating system designed to minimize the amount of code that must be trusted. HiStar provides strict information flow control, which allows users to specify precise data security policies without unduly limiting the structure of applications. HiStar's security features make it possible to implement a Unix-like environment with acceptable performance almost entirely in an untrusted user-level library. The system has no notion of superuser and no fully trusted code other than the kernel. HiStar's features permit several novel applications, including an entirely untrusted login process, separation of data between virtual private networks, and privacy-preserving, untrusted virus scanners.

1 Introduction

Many serious security breaches stem from vulnerabilities in application software. Despite an extensive body of research in preventing, detecting, and mitigating the effects of software bugs, the security of most systems ultimately depends on a large fraction of the code behaving correctly. Unfortunately, experience has shown that only a handful of programmers have the right mindset to write secure code, and few applications have the luxury of being written by such programmers. As a result, we see a steady stream of high-profile security incidents.

How can we build secure systems when we cannot trust programmers to write secure code? One hope is to separate the security critical portions of an application from the untrusted bulk of its implementation; if security depends on only a small amount of code, this code can be verified or implemented by trustworthy parties regardless of the complexity of the application as a whole. Unfortunately, traditional operating systems do not lend themselves to such a division of functionality; they make it too difficult to predict the full implications of every action by untrusted code. HiStar is a new operating system designed to overcome this limitation.

HiStar enforces security by controlling how information flows through the system. Hence, one can reason about which components of a system may affect which others and how, without having to understand those components themselves. Specifying policies in terms of information flow is often much easier than reasoning about the security implications of individual operations.

As an example, consider the recently discovered criti-

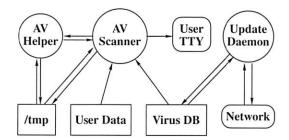


Figure 1: The ClamAV virus scanner. Circles represent processes, rectangles represent files and directories, and rounded rectangles represent devices. Arrows represent the expected data flow for a well-behaved virus scanner.

cal vulnerability in Norton Antivirus that put millions of systems at risk of remote compromise [15]. Suppose we wanted to avoid a similar disaster with the simpler, opensource ClamAV virus scanner. ClamAV is over 40,000 lines of code—large enough that hand-auditing the system to eliminate vulnerabilities would at the very least be an expensive and lengthy process. Yet a virus scanner must periodically be updated on short notice to counter new threats, in which case users would face the unfortunate choice of running either an outdated virus scanner or an unaudited one. A better solution would be for the operating system to enforce security without trusting ClamAV, thereby minimizing potential damage from ClamAV's vulnerabilities.

Figure 1 illustrates ClamAV's components. How can we protect a system should these components be compromised? Among other things, we must ensure a compromised ClamAV cannot purloin private data from the files it scans. In doing so, we must also avoid imposing restrictions that might interfere with ClamAV's proper operation—for example, the scanner needs to spawn a wide variety of external helper programs to decode input files. Here are just a few ways in which, on Linux, a maliciously-controlled scanner and update daemon can collude to copy private data to an attacker's machine:

- The scanner can send the data directly to the destination host over a TCP connection.
- The scanner can arrange for an external program such as sendmail or httpd to transmit the data.
- The scanner can take over an existing process with the ptrace system call or /proc file system, then transmit the data through that process.
- The scanner can write the data to a file in /tmp. The

update daemon can then read the file and leak the data by encoding it in the contents, ordering, or timing of subsequent outbound update queries.

• The scanner can use any number of less efficient and subtler techniques to impart the data to the update daemon—e.g., using system V shared memory or semaphores, calling lockf on various ranges of the database, binding particular TCP or UDP port numbers, modulating memory or disk usage in a detectable way, calling setproctitle to change the output of the ps command, or co-opting some unsuspecting third process such as portmap whose legitimate function can relay information to the update daemon.

Some of these attacks can be mitigated by running the scanner with its own user ID in a *chroot* jail. However, doing so requires highly-privileged, application-specific code to set up the *chroot* environment, and risks breaking the scanner or one of its helper programs due to missing dependencies. Other attacks, such as those involving sockets or System V IPC, can only be prevented by modifying the kernel to restrict certain system calls. Unfortunately, devising an appropriate policy in terms of system call arguments is an error-prone task, which, if incorrectly done, risks leaking private data or interfering with operation of a legitimate scanner.

A better way to specify the desired policy is in terms of where information should flow—namely, along the arrows in the figure. While Linux cannot enforce such a policy, HiStar can. Figure 2 shows our port of ClamAV to HiStar. There are two differences from Linux. First, we have labeled files with private user data as *tainted*. Tainting a file restricts the flow of its contents to any untainted component, including the network. A file can be labeled with arbitrarily many *categories* of taint. Whoever allocates a category—in this case the file owner—has the exclusive ability to *untaint* data in that category.

The second difference from Linux is that we have launched the scanner from a new, 110-line program called *wrap*, to which we give untainting privileges. *wrap* untaints the virus scanner's result and reports back to the user. The scanner cannot read tainted user files without first tainting itself. Once tainted, it can no longer convey information to the network or update daemon. So long as *wrap* is correctly implemented, then, ClamAV cannot leak the contents of the files it scans.

Though this paper will use the virus scanner as a running example, a number of other typical security problems can more easily be couched in terms of information flow. For example, protecting users' private profile information on a web site often boils down to ensuring one person's information (social security number, credit card, etc.) cannot be sent to another user's browser. Protecting against trojan horses means ensuring network payloads do not affect the contents of system

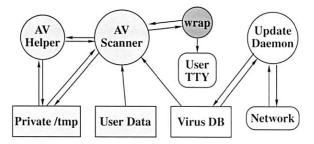


Figure 2: ClamAV running in HiStar. Lightly-shaded components are tainted, which prevents them from conveying any information to untainted (unshaded) components. The strongly-shaded wrap has untainting privileges, allowing it to relay the scanner's output to the terminal.

files. Protecting passwords means ensuring that whatever code verifies them can reveal only the single bit signifying whether or not authentication succeeded. HiStar provides a new, Unix-like development environment in which small amounts of code can secure much larger, untrusted applications by enforcing such policies.

The information flow principles behind this type of isolation are not new. Mechanisms in several other systems, including SELinux [11], EROS [23], and Asbestos [5], are also capable of isolating an untrusted virus scanner. HiStar's taint labels, which originated in Asbestos, have features resembling the languagebased labels in Jif and Jflow [14]. Unlike these systems, though, HiStar shows how to construct conventional operating system abstractions, such as processes, from much lower-level kernel building blocks in which all information flow is explicit. HiStar demonstrates that an operating system can dynamically track information flow through tainting without the taint mechanism itself leaking information. By separating resource revocation from access, HiStar also shows how to eliminate the notion of superuser from an operating system without inhibiting system administration; a HiStar administrator can manage the machine with no special right to untaint, read, or write arbitrary user data.

2 LABELS

HiStar tracks and enforces information flow using Asbestos *labels* [5]. All operating system abstractions are layered on top of six low-level kernel object types described in the next section—threads, address spaces, segments, gates, containers, and devices. Every object has a label. The label specifies, for each category of taint, whether the object has untainting privileges for that category (threads and gates can have such privileges), and, if not, how tainted the object is in that category. Any system call or page fault can cause information to flow between the current thread and other objects. However, the kernel disallows actions that would convey information from more to less tainted objects in any given category.

A label is a function from categories to taint levels.

Level	Meaning in an object's label				
*	has untainting privileges in this category				
0	cannot be written/modified by default				
1	default level—no restriction in this category				
2	cannot be untainted/exported by default				
3	cannot be read/observed by default				

Figure 3: An object's label assigns it one of the above taint levels in each category. Only thread and gate labels may contain \star .

Any given label maps all but a small number of categories to some default background taint level for the object—usually 1. Thus, a label consists of a default taint level and a list of categories in which the object is either more or less tainted than the default. We write labels inside braces, using a comma-separated list of category-level pairs followed by the default level. For example, a typical label might be $L = \{w0, r3, 1\}$, which is just a more compact way of designating the function

$$L(c) = \begin{cases} 0 & \text{if } c = w, \\ 3 & \text{if } c = r, \\ 1 & \text{otherwise.} \end{cases}$$

Each category in which an object's taint differs from the default level 1 places a restriction on how other threads may access the object. To see this, consider a thread T with label $L_T = \{1\}$, and an object O with label $L_O = \{c3, 1\}$. Because $L_T(c) = 1 < 3 = L_O(c)$, O is more tainted than T in category c. Hence, no information may flow from O to T, which means the thread cannot read or observe the object. Conversely, an object may be less tainted than the default. If instead an object O' has $L_{O'} = \{c0, 1\}$, then $L_{O'}(c) = 0 < 1 = L_T(c)$, and no information can flow from T to O', meaning the thread cannot write to or modify the object.

Any given category in an object's label restricts either reading or writing the object, but not both. (It is, of course, common to restrict both by using two categories.) While conventional operating systems can either permit or prohibit read access to an object such as a file, HiStar allows a third option: permit a thread to read an object so long as it does not untaint the data or export it from the machine. In some cases, such as VPN isolation discussed in Section 6.3, it is convenient to make read without untainting the default permission for a given category. Therefore, HiStar supports two levels more tainted than the default: 2 and 3. The difference arises because threads may chose to taint themselves to read more tainted objects, but only up to another label called their *clearance*, which defaults to {2}.

The final taint level is \star ("Star"). It signifies untainting privileges within a category, and may appear only in a thread or gate label. Roughly speaking, when a thread is at level \star in a particular category, the kernel ignores that category in performing label checks for operations

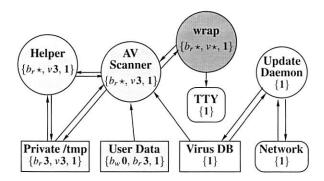


Figure 4: Labels on components of the HiStar ClamAV port.

by that thread. In other words, if a thread T with label L_T has $L_T(c) = \star$, the thread can bypass information flow restrictions in c; we therefore say T owns c. A thread that owns a category can also *grant* ownership of the category to other threads using various mechanisms. Figure 3 summarizes taint levels that appear in object labels.

While there are only a few levels, HiStar supports an effectively unlimited number of categories. Categories are named by 61-bit opaque identifiers, which the kernel generates by encrypting a counter with a block cipher. Encrypting the counter prevents one thread from learning how many categories another thread may have allocated. The counter is sufficiently long that it would take over 60 years to exhaust the identifier space even allocating categories at a rate of one billion per second. Thus, the system permits any thread to allocate arbitrarily many categories. (The specific length 61 was chosen to fit a category name and 3-bit taint level in the same 64-bit field, which facilitated the label implementation.)

A thread that allocates a category is granted ownership of that category. We note this is a significant departure from traditional military systems, which use categories but typically support only a fixed number that must be assigned by the privileged security administrator.

2.1 Example

Returning to the virus scanner example, Figure 4 shows a simplified version of the labels that would arise if a hypothetical user, "Bob," ran ClamAV on HiStar. Before even launching the virus scanner, permissions must be set to restrict access to Bob's files—otherwise, the update daemon could directly read Bob's files and transmit them over the network. In Unix, Bob's files would be protected either by setting file permission bits to 0600 or by running the update daemon in a *chroot* jail. In HiStar, labels can achieve equivalent results.

The equivalent of setting Unix permissions bits is for Bob to allocate two categories, b_r and b_w , to restrict read and write access to his files, respectively. Bob labels his data $\{b_r 3, b_w 0, 1\}$. Threads that own b_r can read the data, so b_r acts like a read capability. Similarly b_w acts like a write capability. The authentication mechanism

described in Section 6.2 grants Bob's shell ownership of the two categories whenever he logs in.

The wrap program is invoked with all of Bob's privileges—in particular with ownership of b_r , the category that restricts read access to Bob's files. wrap allocates a new category, v, to isolate the scanner, creates a private /tmp directory writable at taint level 3 in category v, then launches the scanner tainted 3 in category v. The v taint prevents the scanner, or any process it creates, from communicating to the update daemon or network, except through wrap (which has untainting privileges in v). The v taint also prevents the scanner, or any program it spawns, from modifying any of Bob's files, because those files are all less tainted (at the default level of 1) in category v.

2.2 Notation

Almost every operation in HiStar requires the kernel to check whether information can flow between objects. In the absence of level \star , information can flow from an object labeled L_1 to one labeled L_2 only if L_2 is at least as tainted as L_1 in every category. This relationship is so important that we introduce a symbol, \sqsubseteq , to denote it:

$$L_1 \sqsubseteq L_2$$
 iff $\forall c : L_1(c) \leq L_2(c)$.

Level \star complicates matters since it represents ownership and untainting privileges rather than taint. A thread T whose label L_T maps a category to level \star can ignore information flow constraints on that category when reading or writing objects. When comparing L_T to an object's label, the \star must be considered either less than or greater than numeric levels, depending on context. When T reads an object, \star should be treated as high (greater than any numeric level) to allow observation of arbitrarily tainted information. Conversely, when T writes an object, \star should be treated as low (less than any numeric level) so that information can flow from T to objects at any taint level in the category. This shift from high to low implements untainting.

Rather than have \star take on two possible values in label comparisons, we use two different symbols to represent ownership, depending on context. The existing \star symbol represents the ownership level of a category when it should be treated low. A new \odot ("HiStar") symbol represents the same ownership level when it should be treated high. This gives us a notation with six "levels," ordered $\star < 0 < 1 < 2 < 3 < \odot$. However, level \odot is only used in access rules and never appears in labels of actual objects.

The shifting between levels \star and \odot required for untainting is denoted by superscript operators $^{\circ}$ and * that translate \star to \odot and \odot to \star , respectively. For example, if $L = \{a\star, b\odot, 1\}$, then $L^{\circ} = \{a\odot, b\odot, 1\}$ and $L^{\star} = \{a\star, b\star, 1\}$.

We can now precisely specify the restrictions imposed by HiStar when a thread T labeled L_T attempts to access an object O labeled L_O :

- T can observe O only if $L_O \subseteq L_T^{\circ}$ (i.e., "no read up").
- T can modify O, which in HiStar implies observing O, only if L_T ⊆ L_O ⊆ L^o_T (i.e., "no write down").

These two basic conditions appear repeatedly in our description of HiStar's abstractions.

Labels form a lattice [4] under the partial order of the \sqsubseteq relation. We write $L_1 \sqcup L_2$ to designate the least upper bound of two labels L_1 and L_2 . The label $L = L_1 \sqcup L_2$ is given by $L(c) = \max(L_1(c), L_2(c))$. As previously mentioned, threads may choose to taint themselves to observe more tainted objects. To observe an object O labeled L_O , a thread T labeled L_T must raise its label to at least $L_T' = (L_T'' \sqcup L_O)^*$, because that is the lowest label satisfying both $L_T \sqsubseteq L_T'$ and $L_O \sqsubseteq L_T''$.

3 KERNEL DESIGN

As previously mentioned, the HiStar kernel is organized around six object types. Every object has a unique, 61-bit *object ID*, a *label*, a *quota* bounding its storage usage, 64 bytes of mutable, user-defined *metadata* (used, for instance, to track modification time), and a few *flags*, such as an *immutable* flag that irrevocably makes the object read-only. Except for threads, objects' labels are specified at creation and then immutable. Some objects allow efficient copies to be made with different labels, which is useful in cases that might otherwise require re-labeling.

An object's label controls information flow to and from the object. In particular, the kernel interface was designed to achieve the following property:

The contents of object A can only affect object B if, for every category c in which A is more tainted than B, a thread owning c takes part in the process.

This is a powerful property. It provides end-to-end guarantees of which system components can affect which others without the need to understand either the components or their interactions with the rest of the system.

To revisit the virus scanner example, suppose data from the scanner, tainted v3, was somehow observed by the update daemon, with a label of $\{1\}$. It follows that the *wrap* program—the only owner of v—allowed this to happen in some way, either directly or by pre-authorizing actions on its behalf (for instance, by creating a gate). The privacy of the user's data now depends only on the *wrap* program being correct, and not on the virus scanner. In general, we try to structure applications so that key categories are owned by small amounts of code, and hence the bulk of the system is not security-critical.

Unfortunately, information flow control is not perfect. Tainted malicious software can leak information through

covert channels—for instance, by modulating CPU usage in a way that affects the response time of untainted threads. A related problem is preventing malicious software from making even properly tainted copies of data it cannot read. Such copies could divulge unintended information—for instance, allowing someone who just got ownership of a category to read tainted files that were supposed to have been previously deleted. Restricting copies also lets one limit the amount of time malicious software can spend leaking data over covert channels.

To prevent code from accessing or copying inappropriate data, each thread has a *clearance* label, specifying an upper bound both on the thread's own label and on the labels of objects the thread allocates or grants storage to. In the virus scanner example, the update daemon cannot read Bob's private files, labeled $\{b_r, b_w, 0, 1\}$, because its clearance of $\{2\}$ prevents it from tainting itself $b_r, 3$.

HiStar has a single-level store—on bootup, the entire system state is restored from the most recent ondisk snapshot. This eliminates the need for trusted boot scripts to re-initialize processes such as daemons that on more traditional operating systems would not survive a reboot. It also achieves economy of mechanism by allowing the file system to be implemented with the same kernel abstractions as virtual memory. On the other hand, persistence opens up a host of other issues, chief among them the fact that one can no longer rely on rebooting to kill off errant applications and reclaim resources.

Indeed, resource exhaustion is a potentially troublesome issue for many systems (including Asbestos). The ability to run a machine out of memory is at best a glaring covert channel and at worst a threat to system integrity. HiStar's single-level store at least reduces the problem to disk-space exhaustion, since all kernel objects are written to disk at each snapshot and can be evicted from memory once stably stored. HiStar prevents disk space exhaustion by enforcing object quotas. Quotas form a hierarchy under top-level control of the system administrator—the only inherent hierarchy in HiStar.

The simplest kernel object is a segment, providing a variable-length byte array—similar to a file in other operating systems. The rest of this section discusses other HiStar kernel object types.

3.1 Threads

As previously mentioned, each thread T has a label L_T and a clearance C_T . By default, T has $L_T(c) = 1$ and $C_T(c) = 2$ for most categories c, but the system call

• cat_t create_category (void)

pseudo-randomly chooses a previously unused category, c, and sets $L_T(c) \leftarrow \star$ and $C_T(c) \leftarrow 3$. At that point T is the only thread whose label maps c to a value below the system default of 1. In this sense, labels are egalitarian: no thread has any inherent privileges with respect to

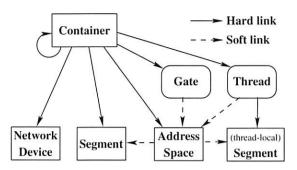


Figure 5: Kernel object types in HiStar. Soft links name objects by a particular (container ID, object ID) container entry. Threads and gates, which can own categories (i.e., contain * in their labels), are represented by rounded rectangles.

categories created by other threads.

T may raise its own label through the system call

• int self_set_label (label_t L),

which sets $L_T \leftarrow L$ so long as $L_T \sqsubseteq L \sqsubseteq C_T$. This can, for example, let T read a tainted object. T can also lower its clearance in any category (but not below its label), or increase its clearance in categories it owns, using

int self_set_clearance (label_t C),

which sets $C_T \leftarrow C$ so long as $L_T \sqsubseteq C \sqsubseteq (C_T \sqcup L_T^{\circ})$.

 L_T and C_T restrict the label L of any object T creates to the range $L_T \sqsubseteq L \sqsubseteq C_T$. Similarly, any new thread T' that T spawns must satisfy $L_T \sqsubseteq L_{T'} \sqsubseteq C_{T'} \sqsubseteq C_T$.

3.2 Containers

Because HiStar has no notion of superuser yet allows any software to create protection domains, nothing prevents a buggy thread from allocating resources in some new, unobservable, unmodifiable protection domain. We must ensure such resources can nonetheless be deallocated.

HiStar provides hierarchical control over object allocation and deallocation through a *container* abstraction. Like Unix directories, containers hold *hard links* to objects. There is a specially-designated root container, which can never be deallocated. Any other object is deallocated once there is no path to it from the root container. Figure 5 shows the possible links between containers and other object types.

When allocating an object, a thread must specify both the container into which to place the object and a 32-byte descriptive string intended to give a rough idea of the object's purpose (much as the Unix *ps* command associates command names with process IDs). For example, to create a container, thread *T* makes the system call

 id_t container_create (id_t D, label_t L, char *descrip, int avoid_types, uint64_t quota).

Here *D* is the object ID of an existing container, into which the newly created container will be placed. (We use *D* for containers to avoid confusion with clearance.) *L* is the desired label for the new container, and *descrip*

is the descriptive string. *avoid_types* is a bitmask specifying kernel object types (e.g., threads) that cannot be created in the container or any of its descendants. *quota* is discussed in the next subsection. The system call succeeds only if T can write to D (i.e., $L_T \sqsubseteq L_D \sqsubseteq L_T^{\bullet}$) and allocate an object of label L (i.e., $L_T \sqsubseteq L \sqsubseteq C_T$).

Objects can be *unreferenced* from container *D* by any thread that can write to *D*. When an object has no more references, the kernel deallocates it. Unreferencing a container causes the kernel to recursively unreference the entire subtree of objects rooted at that container.

HiStar implements directories with containers. By convention, each process knows the container ID of its root directory and can walk the file system by traversing the container hierarchy. The file system uses a separate segment in each directory container to store file names.

A thread T can create a hard link to segment S in container D if it can write D (i.e., $L_T \sqsubseteq L_D \sqsubseteq L_T^{\circ}$) and its clearance is high enough to allocate objects at S's label $(L_S \sqsubseteq C_T)$. T can thus prolong S's life even without permission to modify S. A thread T' must not observe that T has done this, however, unless T could have otherwise communicated to T'—i.e., $L_T \sqsubseteq L_{T'}^{\circ}$ (which need not be the case just because T' has read permission on S). Most system calls therefore specify objects not by ID, but by \langle container ID, object ID \rangle pairs, called *container entries*. For T' to use container entry $\langle D, S \rangle$, D must contain a link to S and T' must be able to read D—i.e., $L_D \sqsubseteq L_{T'}^{\circ}$; since T had $L_T \sqsubseteq L_D$, this implies $L_T \sqsubseteq L_{T'}^{\circ}$, as required.

Container entries let the kernel check that a thread has permission to know of an object's existence. When a thread has this permission, it may also read immutable data specified at the object's creation. In particular, for any object $\langle D,O\rangle$, if T can read D, then T can also read O's descriptive string and, unless O is a thread, O's label. (Since thread labels are not immutable, T can only read the label of another thread T' if $L_{T'}^{\bullet} \sqsubseteq L_{T}^{\bullet}$.) By examining the labels of objects more tainted than themselves, threads can determine how they must taint themselves if they wish to read those objects.

As a special case, every container contains itself. A thread T can access container D as $\langle D, D \rangle$ when $L_D \sqsubseteq L_T^{\circ}$, even if T cannot read D's parent, D'. (The root container has a fake parent labeled $\{3\}$, and must always be referenced this way.) One consequence is that if $L_{D'} \not\sqsubseteq L_D$, a thread with write permission on D' but not D can nonetheless deallocate D in an observable way. By making D less tainted than its parent in one or more categories, the thread T' that created D effectively preauthorized a small amount of information to be transmitted from threads that can delete D to threads that can use D. Fortunately, the allocation rules $(L_{T'} \sqsubseteq L_{D'} \sqsubseteq L_{T'}^{\circ}$ and $L_{T'} \sqsubseteq L_D \sqsubseteq C_{T'}$) imply that to create such a D in D', T' must own every category c for which $L_D(c) < L_{D'}(c)$.

3.3 Quotas

Every object has a *quota*, which is either a limit on its storage usage or the reserved value ∞ (which the root container always has). A container's usage is the sum of the space used by its own data structures and the quotas of all objects it contains. One can adjust quotas with the system call

• int quota_move (id_t D, id_t O, int64_t n),

which adds n bytes to both O's quota and D's usage. D must contain O, and the invoking thread T must satisfy $L_T \sqsubseteq L_D \sqsubseteq L_T^{\circ}$ and $L_T \sqsubseteq L_O \sqsubseteq C_T$. If n < 0, L_T must also satisfy $L_O \sqsubseteq L_T^{\circ}$ because the call returns an error when O has fewer than |n| spare bytes, thereby conveying information about O to T.

Threads and segments can both be hard linked into multiple containers; HiStar conservatively "double-charges" for such objects by adding their entire quota to each container's usage. One cannot add a link to an object whose quota may subsequently change. The kernel enforces this with a "fixed-quota" flag on each object. The flag must be set (though a system call) before adding a link to the object, and can never be cleared.

We do not expect users to manage quotas manually, except at the very top of the hierarchy. The system library can manage quotas automatically, though we do not yet enable this feature by default.

3.4 Address spaces

Every running thread has an associated address space object containing a list of VA $\rightarrow \langle S, offset, npages, flags \rangle$ mappings. VA is a page-aligned virtual address. $S = \langle D, O \rangle$ is a container entry for a segment to be mapped at VA. offset and npages can specify a subset of S to be mapped. flags specifies read, write, and execute permission (and some convenience bits for user-level software).

Each address space A has a label L_A , to which the usual label rules apply. Thread T can modify A only if $L_T \sqsubseteq L_A \sqsubseteq L_T^{\circ}$, and can observe or use A only if $L_A \sqsubseteq L_T^{\circ}$. When launching a new thread, one must specify its address space and entry point. The system call self_set_as also allows threads to switch address spaces. When thread T takes a page fault, the kernel looks up the faulting address in T's address space to find a segment $S = \langle D, O \rangle$ and flags. If flags allows the access mode, the kernel checks that T can read D and $O(L_D \sqsubseteq L_T^{\circ})$ and $L_O \sqsubseteq L_T^{\circ}$). If flags includes writing, the kernel additionally checks that T can modify $O(L_T \sqsubseteq L_O)$. If no mapping is found or any check fails, the kernel calls up to a user-mode page-fault handler (which by default kills the process). If the page-fault handler cannot be invoked, the thread is halted.

Every thread has a one-page local segment that can be mapped in its address space using a reserved object ID meaning "the current thread's local segment." Threadlocal segments are always writable by the current thread. They provide scratch space to use when other parts of the virtual address space may not be writable. For example, when a thread raises its label, it can use the local segment as a temporary stack while creating a copy of its address space with a writable stack and heap.

A system call *thread_alert* allows a thread T' to send an alert to T, which pushes T's registers on an exception stack and vectors T's PC to an alert handler. To succeed, T' must be able to write T's address space A (i.e., $L_{T'} \sqsubseteq L_A \sqsubseteq L_{T'}^{\circ}$) and to observe T (i.e., $L_T \sqsubseteq L_{T'}^{\circ}$). These conditions suffice for T' to gain full control of T by replacing the text segment in A with arbitrary code, as well as for T to communicate information to T'.

3.5 Gates

Gates provide protected control transfer, allowing a thread to jump to a pre-defined entry point in another address space with additional privilege. A gate object G has a gate label, L_G (which may contain \star), a clearance, C_G , and thread state, including the container entry of an address space, an initial entry point, an initial stack pointer, and some closure arguments to pass the entry point function. A thread T' can only allocate a gate G whose label and clearance satisfy $L_{T'} \sqsubseteq L_G \sqsubseteq C_G \sqsubseteq C_{T'}$.

The thread T invoking G must specify a requested label, L_R , and clearance, C_R , to acquire on entry. T also supplies a verify label, L_V , to prove possession of categories without granting them across the gate call. Gate invocation is permitted when $L_T \sqsubseteq C_G$, $L_T \sqsubseteq L_V$, and $(L_T^{\circ} \sqcup L_G^{\circ})^* \sqsubseteq L_R \sqsubseteq C_R \sqsubseteq (C_T \sqcup C_G)$. The entry point function can examine L_V for additional access control. Note that thread labels are always explicitly specified by user code, and only verified by the kernel.

Gates are usually used like an RPC service. Unlike typical RPC, where the RPC server provides the resources to handle the request, gates allow the client to donate initial resources—namely, the thread object which invokes the gate. Arguments and return values are passed across the gate in the thread local segment. Gates can be used to transfer privilege; for example, the login process, described in Section 6.2, uses gates to obtain the user's privileges. The use of gates in user-level applications is discussed in more detail in Section 5.5.

4 KERNEL IMPLEMENTATION

Our implementation of HiStar runs on x86-64 processors, such as AMD Opteron and Athlon64 CPUs. The use of a 64-bit processor makes virtual memory an abundant resource, allowing us to make certain simplifications in our design, such as the use of virtual memory for file descriptors, described in the next section.

The single-level store is inspired by XFS [24]. It uses a B+-tree to store an on-disk mapping from object IDs

to their location on disk, and two B+-trees to maintain a list of free disk space extents. The first one is indexed by extent size and is used to find appropriately-sized extents, and the other is indexed by extent location and is used to coalesce adjacent extents. Our B+-trees have fixed-size keys and values—object IDs and disk offsets—which significantly simplifies their implementation. Write-ahead logging ensures atomicity and crash-consistency. Disk space allocation is delayed until an object is written to disk, making it easier to allocate contiguous extents.

The kernel performs several key optimizations. It caches the result of comparisons between immutable labels. When switching between similar address spaces, it also invalidates TLB entries with the *invlpg* instruction instead of flushing the whole TLB by re-loading the page table base register. The *invlpg* optimization makes switching between threads in the same address space efficient: at worst, the kernel invalidates one page translation for the thread-local segment.

4.1 Code size

One of the advantages of HiStar's simple kernel interface is that the fully-trusted kernel can be quite small. Our kernel implementation consists of 15,200 lines of C code (of which 5,700 lines contain a semicolon) and 150 lines of assembly; this is roughly 45% fewer lines of C code than the Asbestos kernel. The source code consists of the following rough components:

- 3,400 lines of architecture-specific code, implementing virtual memory and threads.
- 4,000 lines of code for B+-trees, write-ahead logging and object persistence.
- 3,000 lines of code for device drivers, including PCI support, DMA-based IDE, console, and three network drivers.
- 4,800 lines of code for system calls, containers, profiling, and other hardware-independent components.

In all aspects of the design we have tried to optimize for a simpler and cleaner kernel. For example, IPC support, aside from shared memory and gates, is limited to a memory-based futex [6] synchronization primitive, on which the user-level library implements mutexes. The kernel network API consists of three system calls: get the MAC address of the card, provide a transmit or receive packet buffer, and wait for a packet to be received or transmitted. There is no dynamic packet allocation or queuing in the kernel, which simplifies drivers. Our DMA-based Intel eepro 100 driver is 500 lines of code, compared to 2,500 in Linux and OpenBSD (not including their in-kernel packet allocation and queuing code). When hardware support for IO virtualization becomes available, we expect to move many device drivers out of the fully-trusted kernel.

5 USER-LEVEL DESIGN

Unix provides a general-purpose computing environment familiar to many people. In designing HiStar's user-level infrastructure, our goal was to provide as similar an environment to Unix as possible except in areas where there were compelling reasons not to—for instance, user authentication, which we redesigned for better security. As a result, porting software to HiStar is relatively straightforward; code that does not interact with security aspects such as user management often requires no modification.

The bulk of the Unix environment is provided by a port of the uClibc library [25] to HiStar. The HiStar platform-specific code is a small layer underneath uClibc that emulates the Linux system call interface, comprising approximately 10,000 lines of code and providing abstractions like file descriptors, processes, fork and exec, file system, and signals. Two additional services—networking and authentication—are provided by separate daemons. A daemon in HiStar is a regular process that creates one or more *service gates* for other processes to communicate with it in an RPC-like fashion.

It is important to note that all of these abstractions are provided at user level, without any special privilege from the kernel. Thus, all information flow, such as the exit status of a child process, is made explicit in the Unix library. A vulnerability in the Unix library, such as a bug in the file system, only compromises threads that trigger the bug—an attacker can only exercise the privileges of the compromised thread, likely causing far less damage than a kernel vulnerability. An untrusted application, such as a virus scanner, can be isolated together with its Unix library, allowing for control over Unix vulnerabilities.

We have ported a number of Unix software packages to HiStar, including GNU coreutils (ls, dd, and so on), ksh, gcc, gdb, the links web browser and OpenSSH, in many cases requiring little or no source code modifications. The rest of this section discusses the design and implementation of our Unix emulation library.

5.1 File System

The HiStar file system uses segments and containers to implement files and directories, respectively. Each file corresponds to a segment object; to access the file contents, the segment is mapped into the thread's address space, and any reads or writes are translated into memory operations. The implementation coordinates with the user-mode page fault handler to return errors rather than SIGSEGV signals upon invalid read or write requests. A file's length is defined to be the segment's length. Extending a file may require increasing the segment's quota, which is done through a gate call if the enclosing container is not writable in the current context. Additional state, such as the modification time, is stored in the object's metadata.

A directory is a container with a special *directory seg- ment* mapping file names to object IDs. Directory operations are synchronized with a mutex in the directory
segment; for example, atomic rename within a directory
is implemented by obtaining the directory's mutex lock,
modifying the directory segment to reflect the new name,
and releasing the lock. Users that cannot write a directory cannot acquire the mutex, but they can still obtain
a consistent view of directory segment entries by atomically reading a generation number and busy flag before
and after reading each entry. The generation number is
incremented by the library on each directory update.

The container ID of the / directory is stored by the Unix library in user space and passed to child processes across fork and exec. The library also maintains a *mount table segment*, which maps \(\langle directory, name \rangle \) pairs onto object IDs. The library overlays mounted objects on directories, much like Unix. Like Plan 9, a process may copy and modify its mount table, for example at user login. The kernel has a *container_get_parent* system call which is used to implement parent directories.

Since file system objects directly correspond to HiStar kernel objects, permissions are specified in terms of labels and are enforced by the kernel, not by the untrusted user-level file system implementation. The label on a file segment is typically $\{r3, w0, 1\}$, where categories r and w represent read and write privilege on that file, respectively. Labels are similarly used for directories; read privilege on a directory allows listing the files in that directory, and write privilege allows creating new files and renaming or deleting existing files.

5.2 Processes

A process in HiStar is a user-space convention. Figure 6 illustrates the kernel objects that make up a typical process; although this may appear complex, it is implemented as untrusted library code that runs only with the privileges of the invoking user.

Each process P has two categories, p_r and p_w , that protect its secrecy and integrity, respectively. Threads in a process typically have a label of $\{p_r \star, p_w \star, 1\}$, granting them full access to the process. The process consists of two containers: a process container and an internal container. The process container exposes objects that define the external interface to the process: a gate for sending signals and a segment to store the process's exit status; not shown is a gate used by gdb for debugging. The process container and exit status segment are labeled $\{p_w 0, 1\}$, allowing read but not write access by threads of other processes (which do not own p_w). The signal gate has label $\{p_r \star, p_w \star, 1\}$ and allows other processes to send signals to this process. The internal container, address space, and segment objects are labeled $\{p_r, p_w, 0, 1\}$, preventing direct access by other processes.

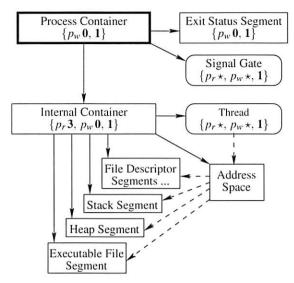


Figure 6: Structure of a HiStar process. A process container is represented by a thick border. Not shown are some label components that prevent other users from signaling the process or reading its exit status.

5.3 File Descriptors

File descriptors in HiStar are implemented in the userspace Unix library. All of the state typically associated with the file descriptor, such as the current seek position and open flags, is stored in a *file descriptor segment*. Every file descriptor number corresponds to a specific virtual memory address. When a file descriptor is open in a process, the corresponding file descriptor segment is memory-mapped at the virtual address for that file descriptor number.

Typically each file descriptor segment has a label of $\{f_r 3, f_w 0, 1\}$, where categories f_r and f_w grant read and write access to the file descriptor state. Access to the descriptor can be granted by setting a thread's label to $\{f_r \star, f_w \star, 1\}$. Multiple processes can share file descriptors by mapping the same descriptor segment into their respective address spaces. By convention, every process adds hard links for all of its file descriptor segments to its own container. As a result, ownership of the file descriptor is shared by all processes holding it open, and a shared descriptor segment is only deallocated when it has been closed and unreferenced by every process.

5.4 Users

A pair of unique categories u_r and u_w define the read and write privileges of each Unix user u in HiStar, including root. Typically, threads running on behalf of user U have a label containing $u_r \star$, $u_w \star$, and users' private files would have a label of $\{u_r 3, u_w 0, 1\}$. One consequence of this design is that a single process can possess the privilege of multiple users, or perhaps multiple user roles, something hard to implement in Unix. On the other hand, our prototype does not support access control lists. (Doing so would probably require a gate for

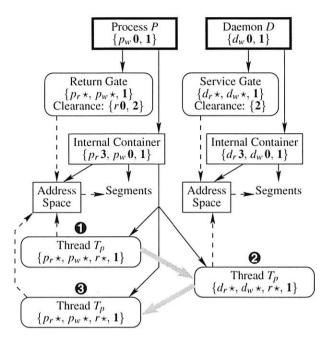


Figure 7: Objects involved in a gate call operation. Thick borders represent process containers. r is the return category; d_r and d_w are the process read and write categories for daemon D. Three states of the same thread object T_p are shown: 1) just before calling the service gate, 2) after calling the service gate, and 3) after calling the return gate.

every access control group.) The authentication service, which verifies user passwords and grants user privileges, is described in more detail in Section 6.2.

5.5 Gate Calls

Gates provide a mechanism for implementing IPC. As an example, consider a service that generates timestamped signatures on client-provided data; such a service could be used to prove possession of data at a particular time. A HiStar process could provide such a service by creating a service gate whose initial entry point is a function that computes a timestamped signature of the input data (from the thread-local segment) and returns the result to the caller. Gates in HiStar have no implicit return mechanism; the caller explicitly creates a return gate before invoking the service gate, which allows the calling thread to regain all of the privileges it had prior to calling the service. A return category r is allocated to prevent arbitrary threads from invoking the return gate; the return gate's clearance requires ownership of the return category to invoke it, and the caller grants the return category when invoking the service gate. Figure 7 shows such a gate call from process P to daemon D.

Suppose the caller does not trust the signaturegenerating daemon D to keep the input data private. To ensure privacy, the calling thread can allocate a new taint category t and invoke the service gate with a label of $\{d_r \star, d_w \star, r \star, t \, 3, 1\}$ —in other words, tainted in the new category. A thread running with this label in D's address space can read any of D's segments, but not modify them (which would violate information flow constraints in category t). However, the tainted thread can make a tainted, and therefore writable, copy of the address space and its segments and continue executing there, effectively forking D into an untainted parent daemon and a tainted child. Unable to divulge the caller's data, the thread can still compute a signature and return it to the caller. Upon invoking the return gate, the thread regains ownership of category t, allowing it to untaint the computed signature.

Resources for the tainted child copy must be charged against some object's quota. They cannot be charged to D's container, because the thread lacks modification permission when tainted t 3 (otherwise, it could leak information about the caller's private data to D). Therefore, before invoking the gate, the calling thread creates a container it can use once inside D. In this example, T_p creates a container labeled $\{t$ 3, r 0, 1 $\}$ inside P's internal container.

Forking on tainted gate invocation is not appropriate for every service. Stateless services such as the timestamping daemon are usually well-suited to forking, whereas services that maintain mutable shared state may want to avoid forking by refusing tainted gate calls.

5.6 Signals

Signals are implemented by sending an alert to a thread in a process, passing the signal number as an argument to the alert handler. The alert handler invokes the appropriate Unix signal handler for the raised signal. However, sending an alert requires the ability to modify the thread's address space object, which, because of p_w , only other threads in the same process can do. Therefore, to support Unix signals, each process exposes a signal gate in its process container. The gate has a label of $\{p_r \star, p_w \star, 1\}$ and an entry function that sends the appropriate alert to one of the threads in the process, depending on the requested signal number. The clearance on the signal gate is $\{u_w 0, 2\}$, where u_w corresponds to the user that is running this process. As a result, only threads that possess the user's privilege can send signals to that user's processes.

5.7 Networking

HiStar uses the lwIP [12] protocol stack to provide TCP/IP networking. lwIP runs in a separate *netd* process and exposes a single gate that allows callers to perform socket operations. Operations on socket file descriptors are translated into gate calls to the netd process. By default, netd's process container is mounted as */netd* in mount tables. As an optimization, a process can create a shared memory segment with netd and donate resources for a worker thread to netd. Subsequent netd interactions can then use futexes to communicate over shared memory, avoiding the overhead of gate calls.

The network device is typically labeled $\{n_r 3, n_w 0, i2, 1\}$, where n_r and n_w are owned by netd, and i taints all data read from the network. Because netd cannot bypass the tainting with i or leak tainted data in other categories, it is mostly untrusted. A compromised netd can only mount the equivalent of a network eavesdropping or packet tampering attack.

5.8 Explicit Information Leaks

Unix was not designed to control information flow. Emulating certain aspects therefore requires information leaks. HiStar implements these leaks at user level, through explicit *untainting gates*. By convention, when spawning a tainted thread or tainting a thread through a gate call, user code supplies the tainted thread with the container entry of an untainting gate. The new thread can invoke this gate to leak certain kinds of information, such as the fact it is about to exit (so the parent shell can reclaim resources and return to the command prompt). Not all categories have untainting gates; whether or not to create one is up to the category's owner.

Currently our Unix library provides untainting gates for up to three operations: process exit, quota adjustment, and file creation. Of these, file creation has by far the biggest information flow, declassifying the name of the newly created file. Low-secrecy applications concerned only with accidental disclosure allow these operations. Higher-secrecy applications may choose to set fixed quotas for tainted objects and only declassify process exits. The next section shows examples of such applications.

6 APPLICATIONS

The Unix environment described in the previous section allows for general-purpose computing on HiStar, but does not provide any functionality qualitatively different from Linux. HiStar's key advantage is that it enables novel, high-security applications to run alongside a familiar Unix environment. This section presents some applications that take advantage of HiStar to provide security guarantees not achievable on typical Unix systems.

6.1 Anti-Virus Software

We have implemented an untrusted virus scanner, as suggested in several examples, by porting ClamAV [3] and using the *wrap* program to run it in isolation. To provide strong isolation, *wrap* does not create the standard Unix untainting gate for category *v. wrap* also limits the amount of data that can be leaked through covert channels by killing ClamAV after some period of time.

ClamAV and its database must be periodically updated to keep up with new viruses. In HiStar, the update process runs with the privilege to write the ClamAV executable and virus database; however, it cannot access private user data. Even if a compromised update installs ar-

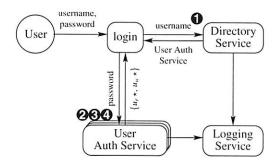


Figure 8: A high-level overview of the authentication system.

bitrary code in place of ClamAV, the label set by *wrap* when running ClamAV ensures that private information cannot be exported.

6.2 User Authentication

User authentication provides a good example of how Hi-Star can minimize trusted code. Most operating systems require a highly-trusted process to validate authentication requests and grant credentials. For example, the Unix login program runs as superuser to set the appropriate user and group IDs after checking passwords. Even a privilege-separated server such as OpenSSH requires a superuser component to be able to launch shells for successfully authenticated users.

In contrast, HiStar authenticates users without any highly-trusted processes, and allows users to supply their own authentication services. Even if a user accidentally provides his or her password to a malicious authentication service, HiStar ensures that only one bit of information about the user's password is leaked. Providing such isolation under a traditional operating system would be difficult.

Figure 8 shows an overview of the HiStar authentication facility. Logically, four entities coordinate to authenticate a user: a login client, a directory service, a per-user authentication service, and a logging service. Of these, the logging service is simplest; the directory and user authentication services trust it to maintain an append-only log, while it trusts them not to exhaust space with spurious entries.

The login client initiates authentication. It typically consists of an instance of the web server or *sshd* that knows a username and password and wishes to gain ownership of the user's read and write categories, u_r and u_w . Login minimally trusts the directory to interpret the username properly (without which authentication could fail or return the wrong credentials). However, login does not trust the other components, and importantly does not trust anyone with the user's password. Conversely, no other component trusts login until it authenticates itself.

The directory service maintains a list of user accounts. Its job is to map usernames to user authentication service daemons. Login begins the authentication process

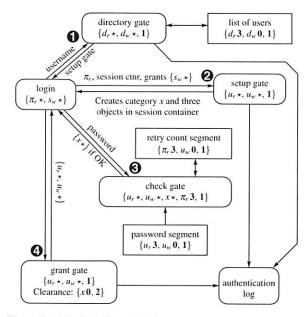


Figure 9: A detailed view of the interactions between authentication system components. The setup gate, check gate and grant gate (2, 3 and 4) are all part of the user's authentication service.

by asking the directory for a particular username. The directory responds with the container entry of a gate to the user's authentication service. The directory is controlled by the system administrator, but is untrusted except minimally by login and the logger as described above.

Each user runs an authentication service daemon that owns u_r and u_w ; the daemon's job is to grant those categories to login clients that successfully authenticate themselves. Conceptually, this is simple: login sends the password to the authentication service, which checks it and, if correct, grants u_r and u_w back to login. Since the authentication service is under the user's control, it can, at the user's option, support non-password techniques such challenge-response authentication.

The complication is that login does not trust the authentication service with the user's password. After all, a mistyped username or malicious directory could connect login to the wrong authentication service. Even the right service might be compromised, which should reveal only the user's password hash, not his password. With challenge-response authentication, a similar manin-the-middle threat exists. The solution is for login to invoke the authentication service three times: first to set things up, second to check the password, and third to finally gain ownership of u_r and u_w . The second step runs tainted, thereby protecting the secrecy of the password.

Figure 9 shows the authentication sequence in more detail. In Step 1, login learns of the appropriate user's setup gate from the directory service. Then it allocates two categories: π_r , the password read category, protects the password from disclosure. The s_w category controls write access to a *login session container*, which login

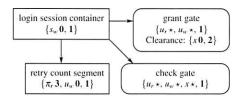


Figure 10: Objects created by the user's setup gate in the session container.

creates with label $\{s_w \mathbf{0}, \mathbf{1}\}.$

In Step 2, login invokes the user's setup gate, granting the user's code $s_w \star$. The setup gate logs the authentication attempt and allocates a new category, x, to be granted to login after successful authentication. Before returning, the setup gate code (together with login, as we will discuss later) creates three objects in the session container, shown in Figure 10. The first is a *retry count segment*, used to bound the number of password guesses per logged invocation of the setup gate. The second is an ephemeral *check gate*, used to check passwords while tainted; its closure arguments specify the object ID of the retry count segment. The third is an ephemeral *grant gate* with clearance $\{x0, 2\}$.

In Step 3, login calls the check gate with the password, tainting the thread $\pi_r 3$. If the password is correct and the retry count okay, the gate code grants x back to login. (Optionally, the check gate may accept a verify label of $\{root_w 0, 3\}$ instead of a password, to emulate a Unix users' trust of root.) Once login owns x, it calls the grant gate in Step 4 to obtain u_r and u_w . The grant gate logs the authentication success before returning, which is why it must be separate from the tainted check gate, which cannot talk to the logging service.

In Step 2, creating the retry count segment, which is labeled $\{\pi_r 3, u_w 0, 1\}$, requires combining the privileges of two mutually-distrustful entities: login, with a clearance of $\pi_r 3$, and the user's code, with a label of $u_w \star$. The user's code will not grant $u_w \star$ to login before a successful authentication. Similarly, login does not trust the user's setup gate code with a clearance of $\pi_r 3$.

To see why login cannot invoke the setup gate with a clearance of $\pi_r 3$, consider what malicious setup gate code can do given such a clearance: It can create a long-lived segment S labeled $\{\pi_r 3, u_r 3, 1\}$, and a long-lived thread T labeled $\{\pi_r 3, u_r *, 1\}$. Both can be in a container inaccessible to login. The setup code can furthermore point the check gate to a "trojaned" variant of the password checker that writes the password to S. Finally, T can read S and leak the password through a covert channel over a long period of time. T and S will persist long after login has destroyed all objects it knows about with a clearance of $\pi_r 3$.

To solve this problem, the developers of the user's authentication service and the login client agree ahead of time on a function that both of them want to execute to create the retry count segment. Then, before invoking the setup gate, login creates a code segment containing the code of the previously agreed-upon function, as well as a gate G that invokes this code with a clearance of π_r 3. Additionally, login marks the code segment and address space objects invoked by G as *immutable* in the kernel. Because these objects are immutable, the user's setup gate code can verify their contents and be assured that invoking G with u_w * will execute only the agreed upon code and not somehow result in login usurping ownership of u_w . In this manner, two mutually-distrustful parties can safely execute mutually agreed-upon code with their combined privilege.

The authentication service implementation is fairly small. The logging service comprises 58 lines of code; the directory service comprises 188 lines, and the standard password-based user authentication service comprises 233 lines of code. Common library code that allows combining privileges to create the retry count segment is 370 lines of C++ code, and the mutually agreed-upon code to create the retry count segment is 30 lines of assembly. Aside from security, another advantage of privilege-separating authentication is that the processes can keep relatively small labels, improving the performance of label operations.

6.3 VPN Isolation

Many networks rely so heavily on firewalls for security that the prospect of bridging them to the open Internet poses a serious danger. Indeed, this is how the Slammer worm disabled a safety monitoring system at a nuclear power plant in 2003 [19]. At the same time, it has become quite common for people to connect home machines and laptops to otherwise firewalled networks through encrypted virtual private networks (VPNs). When VPNs let the same machine connect to either side of a firewall, they risk having malware either infect internal machines or (as the Sircam worm did) divulge sensitive documents to the world.

In HiStar, however, one can track the provenance of data with labels and precisely control what flows between networks. The bootstrap procedure already labels the network device to taint anything received from the Internet $\{i\mathbf{2},\mathbf{1}\}$ and block from transmission anything more tainted. One can analogously label all VPN input $\{v\mathbf{2},\mathbf{1}\}$ and block any more tainted VPN output. Such a configuration completely isolates the two networks from each other except as specifically permitted by the owners of i and v. For example, users might be allowed to untaint i (meaning import external data) when the file passes a virus checker, such as the one in Section 6.1.

We have implemented VPN isolation around the popular OpenVPN package [16]. Figure 11 shows the components of the system and their labels: The VPN runs a second lwIP stack which talks to the OpenVPN client

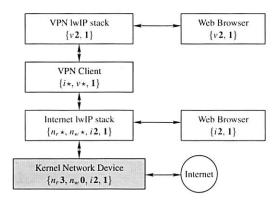


Figure 11: Secure VPN application. The VPN client is trusted to taint incoming VPN packets with $\{v2\}$, reject any outgoing packets tainted in category i, and properly encrypt/decrypt data. The kernel network device is completely trusted. Neither of the lwIP stacks is trusted.

over a *tun* device. Porting OpenVPN to HiStar required implementing a tun character device in the file system library (200 lines of code) and a tun "device driver" for lwIP (100 lines of code). OpenVPN swaps between *v* and *i* taints on the data it encrypts. Users select which network to use by mounting the appropriate lwIP process on */netd* (much like Plan 9). Not shown are untainting gates, which for this application allow processes to leak exit, quota, and file creation events, as discussed in Section 5.8.

VPN isolation is interesting because it applies a broad policy potentially affecting most processes in the system, yet requires only a localized change. This would be difficult to achieve in a capability-based system, for instance.

6.4 Web Services

The original motivating application for Asbestos was its web server, which isolated different user's data to tolerate buggy or malicious web service code. We have built a similar web server for HiStar, with a few differences. HiStar's connection demultiplexer controls resources granted to each worker daemon through containers. Authentication uses an instance of the daemon described in Section 6.2. HiStar also has an experimental privilege-separated database; unlike the Asbestos database, it does not support standard SQL queries. (Whether it will prove general enough for most web services is still an open question.) Since the benefits of Asbestos-style web services have been reported elsewhere, this paper concentrates on other applications whose architecture is more unique to HiStar.

7 PERFORMANCE

To evaluate the performance implications of HiStar's architecture, we compared it to Linux and OpenBSD under several benchmarks. The benchmarks ran on three identical systems, each with a 2.4 GHz AMD Athlon64 3400+ processor, 1GB of main memory, and a 40 GB, 7,200 RPM Seagate ST340014A EIDE hard drive. The

Benchmark	HiStar	Linux	OpenBSD
IPC benchmark, per RTT	3.11 μsec	4.32 μsec	2.13 μsec
Fork/exec, per iteration	1.35 msec	0.18 msec	0.18 msec
Fork/exec, dynamic linking	-	0.45 msec	0.38 msec
Spawn, per iteration	0.47 msec	_	_
LFS small, create, async	0.31 sec	0.316 sec	0.22 sec
per-file sync	459 sec	558 sec	_
group sync	2.57 sec	_	_
LFS small, read, cached	0.16 sec	0.068 sec	0.14 sec
uncached	6.49 sec	1.86 sec	_
no IDE disk prefetch	86.4 sec	86.6 sec	
LFS small, unlink, async	0.090 sec	0.244 sec	0.068 sec
per-file sync	456 sec	173 sec	_
group sync	0.38 sec	_	_
LFS large, sequential write	2.14 sec	3.88 sec	_
sync random write	93.0 sec	89.7 sec	_
LFS large, uncached read	1.96 sec	1.80 sec	_

Figure 12: Microbenchmark results on HiStar, Linux and OpenBSD.

first machine ran HiStar; the second ran Fedora Core 5 Linux with kernel version 2.6.16-1.2080.FC5 x86_64 and an ext3 file system; the third ran 32-bit OpenBSD 3.9 i386 with an in-memory *mfs* file system—a 64-bit version of OpenBSD 3.8 for amd64 performed strictly worse in every benchmark. We did not run synchronous file system benchmarks under OpenBSD, because we could not disable IDE write caching.

7.1 Microbenchmarks

To evaluate the performance of specific aspects of Hi-Star, we chose four microbenchmarks: LFS small-file and large-file benchmarks [20], an IPC benchmark which measures the latency of communication over a Unix pipe, and a fork/exec benchmark that measures the latency of executing /bin/true using fork and exec. All microbenchmarks and /bin/true were compiled statically to eliminate dynamic linking overhead. Figure 12 shows the performance of the four microbenchmarks on three different operating systems.

For the IPC benchmark, two processes are created, connected by two uni-directional pipes; each process sends any messages it receives back to the other process. The benchmark measures the average round-trip time taken to transmit an 8-byte message, over one million round-trips. HiStar performs better than Linux in this benchmark, but somewhat slower than OpenBSD.

HiStar's performance noticeably suffers in the fork and exec microbenchmark. In part, this is because Linux and OpenBSD pre-zero memory pages, which HiStar does not yet do. Moreover, while OpenBSD and Linux require 9 system calls to fork a child, have the child execute /bin/true, have /bin/true exit, and have the parent wait for the child, the same workload requires 317 system calls on top of HiStar's lower-level interface. However, the flexibility provided by a lower-level interface allows us to implement more efficient library calls, such as *spawn*, which directly starts a new process run-

ning a specified executable. The *spawn* function runs 3 times faster than the equivalent fork and exec combination, issuing only 127 system calls per iteration. We note that use of dynamic linking would reduce the relative performance difference between HiStar and Linux.

The LFS small file benchmark creates, reads, and unlinks 10,000 1kB-sized files and reports the total running time for each of these three phases. We measured different variations of the phases, as shown in Figure 12. The asynchronous and cached variations show HiStar has comparable performance to the other systems for requests that go to cache. The uncached read phase measures the time to read 10,000 small files from disk. Here Linux significantly outperforms HiStar, averaging less than 1/10th the disk's 8.3 msec rotational latency to read each file. We attribute this performance to read lookahead in the IDE disk [22], because Linux clusters files from the same directory while HiStar does not. Disabling lookahead, HiStar and Linux perform comparably.

In the synchronous unlink phase, HiStar performs significantly worse than Linux. This is because we implement *fsync* of a directory by checkpointing the entire system state to disk, whereas Linux only writes out the modified directory entry. Synchronous file creation in HiStar also checkpoints the entire system state; however, its performance is comparable to Linux because ext3 performs more writes in this case. Write-ahead logging allows HiStar to achieve acceptable *fsync* performance by queuing updates in a sequential on-disk log. Logged updates are applied in batches; during each run of the synchronous small file benchmarks, the contents of the on-disk log were applied to disk about 10 times (once for approximately every 1,000 synchronous operations).

The single-level store offers a new *group sync* consistency choice not possible under Linux. In group sync, the system state is checkpointed to disk only once at the end of each benchmark phase. The single-level store guarantees that the application either runs to completion or appears never to have started. Using group sync in HiStar, some applications may achieve a significant speedup over Linux, as high as a factor of 200 for applications similar to the LFS small file benchmark.

For the LFS large file benchmark, we evaluated three phases. In the first phase, a 100MB file was created by sequentially writing 8KB chunks, with a single call to fsync at the end of the phase. HiStar achieves close to the maximum disk bandwidth of 58MB/sec [22]; we suspect that block-based (rather than extent-based) allocation in ext3 accounts for Linux's slightly lower performance.

The second phase tested random write throughput; 100MB worth of 8KB chunks were written to random locations in the existing file, and the modifications were *fsynced* to disk for each 8KB write. In the case of pre-existing segments, HiStar allows modified segment

Benchmark	HiStar	Linux	OpenBSD
Building HiStar kernel	6.2 sec	4.7 sec	6.0 sec
Transferring 100MB with wget	9.1 sec	9.0 sec	9.0 sec
Virus-checking a 100MB file	18.7 sec	18.7 sec	21.2 sec
with isolation wrapper	18.7 sec	_	_

Figure 13: Application-level benchmark results.

pages to be flushed to disk (modified in-place) without checkpointing the entire system state. As a result, the performance is again quite close to that of Linux, since each random write involves flushing two 4KB pages to disk both in Linux and in HiStar.

The third phase of the large-file benchmark tested read performance by sequentially reading the 100MB file in 8KB chunks. The performance is approximately the same between HiStar and Linux. Currently the HiStar prototype does not support paging in of partial segments, so the entire 100MB file segment is paged in when the file is first accessed—a limitation we plan to address in the future. As a result, the performance of random reads differs little from the sequential case.

7.2 Application Performance

For an application-level benchmark, we built the HiStar kernel using GNU make 3.80 and GCC 3.4.5 on the three operating systems; Figure 13 summarizes the results. Hi-Star is somewhat slower than Linux and comparable to OpenBSD. In HiStar, most of the CPU time in this benchmark is spent in user space. Since most of our optimization efforts to date have focused on the kernel, we expect HiStar to improve on this benchmark as we move to optimizing the Unix library.

HiStar also achieves good network throughput. When downloading a 100MB file using *wget*, the results show all three operating systems could saturate a 100Mbps Ethernet. Finally, we measured the time taken to check a 100MB file containing randomized binary data for viruses using ClamAV; HiStar performs competitively with Linux and OpenBSD, both with and without the use of the wrapper described in Section 6.1.

8 RELATED WORK

HiStar was directly inspired by Asbestos, but differs in providing system-wide persistence, explicit resource allocation, and a lower-level kernel interface that closes known covert storage channels. While Asbestos is a message-passing system, HiStar relies heavily on shared memory. The HiStar kernel provides gates, not IPC, with the important distinction that upon crossing a gate, a thread's resources initially come from its previous domain. By contrast, Asbestos changes a process's label to track information flow when it receives IPCs, which is detectable by third parties and can leak information. Asbestos highly optimizes comparisons between enormous labels, which so far we have not done in HiStar.

HiStar controls information flow with mandatory access control (MAC), a well-studied technique dating back decades [1]. The ADEPT-50 dynamically adjusted labels (essentially taint tracking) using the High-Water-Mark security model back in the late 1960s [10]; the idea has often resurfaced, for instance in IX [13] and LO-MAC [7]. HiStar and its predecessor Asbestos are novel in that they make operations such as category allocation and untainting available to application programmers, where previous OSes reserved this functionality for security administrators. Decentralized untainting allows novel uses of categories that we believe promote better application structure and support applications, such as web services, not targeted by previous MAC systems.

Superficially, HiStar resembles capability-based KeyKOS [2] and its successor EROS [23]. Both systems use a small number of kernel object types and a single-level store. HiStar's container abstraction is reminiscent of hierarchical space banks in KeyKOS. However, while KeyKOS uses kernel-level capabilities to enforce labels at user-level, HiStar bases all protection on kernel-level labels. The difference is significant because labels specify security properties while imposing less structure on applications—for example, an untrusted thread can dynamically alter its label to observe secret data, which has no analogue in a capability system.

HiStar has no superuser. A number of previous systems have limited, partitioned [13], or virtualized [18] superuser privileges. Several operating systems including Linux support POSIX capabilities, which can permit some superuser privileges while disabling others.

Plan 9 [17] also has no superuser. Administrative tasks such as adding users can only be performed on the file server console, virtually eliminating the threat of network break-ins. On workstations, however, the console user has special privileges, and on compute servers a pseudo-user named "bootes" does. Plan 9 provides a complete, working system with a trusted computing base many times smaller than comparable operating systems. It also provides per-process file namespaces, which inspired HiStar's user-level mount table segments. However, Plan 9 was never intended to support MAC.

HiStar uses gates for protected control transfer, an idea dating back to Multics [21]. However, HiStar's protection domains are not hierarchical like Multics rings. HiStar gates are more like doors in Spring [8].

Decentralized untainting, while new in operating systems, was previously provided by programming languages, notably Jif [14]. There are significant differences between a language and an operating system. Jif can track information flow at the level of individual variables and perform most label checks at compile time. It also has the luxury of relying on the underlying operating system for storage, trusted input files, administration,

etc., which avoids many issues HiStar needs to address.

Singularity [9] provides programming-languagebased security without an underlying operating system. Somewhat like containers, Singularity addresses coherent resource deallocation with a new abstraction called Software-Isolated Processes (SIPs). Singularity does not provide MAC, however.

SELinux [11] lets Linux support MAC; like most MAC systems, policy is centrally specified by the administrator. In contrast, HiStar lets applications craft policies around their own categories of information. Retrofitting MAC to a large existing kernel such as Linux is potentially error-prone, particularly given the sometimes ill-specified semantics of Linux system calls. HiStar's disciplined, small kernel can potentially achieve much higher assurance at the cost of compatibility.

9 LIMITATIONS

We believe HiStar provides a good environment to develop secure applications with small trusted code size. Nonetheless, the system has limitations both in terms of functionality and security. Some of these limitations are artifacts of the implementation that we hope to correct, while others are more fundamental to the approach.

Users familiar with Unix will find that, though HiStar resembles Unix, it also lacks several useful features and changes the semantics of some operations. For example, HiStar does not currently keep file access times; although possible to implement for some cases, correctly tracking time of last access is in many situations fundamentally at odds with information flow control.

Another difference is that *chmod*, *chown*, and *chgrp* revoke all open file descriptors and copy the file or directory. Because each file has one read and one write category, group permissions require a file's owner to be in the group. There is no file execute permission without read permission, and no setuid bit (though gates arguably provide a better alternative to both). Several other facilities are missing, though we hope to add them, including support for system-wide backup and restore, and a user-level trampoline mechanism to allow upgrading of software behind gates (since gate entries are fixed).

Though HiStar is intended to allow administration without a superuser, we do not yet have experience administering a production HiStar system. However, we believe that to the extent it is needed, superuser privilege should be implemented by *convention*—explicitly granting most privilege to the root user—not by *design*. A HiStar administrator can still revoke all resources by virtue of having write permission on the root container. This provides a worst-case answer to uncooperative users that refuse to grant the necessary privilege to root.

While the HiStar kernel provides consistency across kernel crashes and restarts, a crashed or killed process can leave locked mutexes, such as the directory segment mutex. We currently do not recover from such problems, but foresee two potential solutions. The first is to do write-ahead logging in memory; given some way of detecting a dead or crashed process—for example, through timeouts—other processes can recover the directory segment. The second is to prevent the thread from being killed while it is holding the directory mutex, by adding a hard-link to it in the directory container. If the thread is unreferenced from other containers, it will continue executing until removing itself from the directory container.

Because Asbestos labels are more general than capabilities, they allow multiple objects to be protected by the same category and multiple categories to place restrictions on the same object. Users familiar with capability systems will rightfully object that protecting multiple objects with the same category limits the granularity at which privileges can be enumerated. HiStar can be used like a capability system by allocating a new category pair for every object, but our Unix library does not do this. However, as the VPN example showed, HiStar has the advantage of allowing new policies to be overlaid on existing software, which cannot be done as easily in pure capability systems.

One security limitation is that HiStar does not support CPU quotas, though we hope to add these using the container hierarchy. A more serious problem we do not know how to solve is covert timing channels. Many network services have to offer low response latency, and as a result, it becomes increasingly practical to leak information to outside observers by modulating response time.

10 SUMMARY

HiStar is a new operating system that provides strict information flow control without superuser privilege. Narrow interfaces allow for a small trusted kernel of less than 16,000 lines, on which a Unix-like environment is implemented mostly as untrusted user-level library code. A new container abstraction lets administrators manage and revoke resources for processes they cannot observe. Side-by-side with the Unix environment, the system supports a number of high-security, privilege-separated applications previously not possible in a traditional Unix system. Benchmarks show HiStar performs competitively with Linux and OpenBSD.

ACKNOWLEDGMENTS

We thank Hector Garcia-Molina, Michael Freedman, Ramesh Chandra, Constantine Sapuntzakis, Jim Chow, the anonymous reviewers, and our shepherd, Rob Pike, for their feedback. We also thank the Asbestos group, especially Steve VanDeBogart and Petros Efstathopoulos, who co-designed Asbestos labels. HiStar was in part inspired by some of the input Cliff Frey had to the Asbestos

project. This work was funded by joint DARPA/NSF Cybertrust grant CNS-0430425.

REFERENCES

- D. E. Bell and L. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, March 1976.
- [2] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pages 95–112, April 1992.
- [3] ClamAV. http://www.clamav.net/.
- [4] D. E. Denning. A lattice model of secure information flow. Communications of the ACM, 19(5):236–243, May 1976.
- [5] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proc. of the 20th SOSP*, pages 17–30, October 2005.
- [6] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. Ottawa Linux Symposium, 2002.
- [7] T. Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *Proc. of the 2000 IEEE Symposium on Security and Privacy*, pages 230–245, Oakland, CA, May 2000.
- [8] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In Proc. of the Summer 1993 USENIX, pages 147–159, April 1993.
- [9] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft, Redmond, WA, October 2005.
- [10] C. E. Landwehr. Formal models for computer security. Computing Survels, 13(3):247–278, September 1981.
- [11] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. of the 2001 USENIX*, pages 29–40, June 2001. FREENIX track.
- [12] LWIP. http://savannah.nongnu.org/projects/lwip/.
- [13] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. Software—Practice and Experience, 22(8):673–694, 1992.
- [14] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *Transactions on Computer Systems*, 9(4):410–442, October 2000.
- [15] R. Naraine. Symantec antivirus worm hole puts millions at risk. eWeek.com, May 2006. http://www.eweek.com/article2/ 0,1895,1967941,00.asp.
- [16] OpenVPN. http://openvpn.net/.
- [17] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3): 221–254, Summer 1995.
- [18] H. Pötzl. Linux-VServer Technology, 2004. http:// linux-vserver.org/Linux-VServer-Paper.
- [19] K. Poulsen. Slammer worm crashed Ohio nuke plant net. The Register, August 20, 2003. http://www.theregister.co.uk/2003/08/20/slammer_worm_crashed_ohio_nuke/.
- [20] M. Rosenblum and J. Ousterhout. The design and implementation of a logstructured file system. In *Proc. of the 13th SOSP*, pages 1–15, Oct. 1991.
- [21] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. In *Proc. of the Third Symposium on Operating Systems Principles*, pages 42–54, March 1972.
- [22] Seagate. Barracuda 7200.7 Product Manual, Publication 100217279, Rev. L edition, March 2004. http://www.seagate.com/support/disc/manuals/ata/cuda7200pm.pdf.
- [23] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In Proc. of the 17th SOSP, pages 170–185, December 1999.
- [24] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, CA, USA, 22–26 1996.
- [25] uClibc. http://uclibc.org/.

Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable

Richard Ta-Min Lionel Litty* David Lie
Department of Electrical and Computer Engineering
University of Toronto
{tamin,llitty,lie}@eecg.toronto.edu

Abstract

In current commodity systems, applications have no way of limiting their trust in the underlying operating system (OS), leaving them at the complete mercy of an attacker who gains control over the OS. In this work, we describe the design and implementation of Proxos, a system that allows applications to configure their trust in the OS by partitioning the system call interface into trusted and untrusted components. System call routing rules that indicate which system calls are to be handled by the untrusted commodity OS, and which are to be handled by a trusted private OS, are specified by the application developer. We find that rather than defining a new system call interface, routing system calls of an existing interface allows applications currently targeted towards commodity operating systems to isolate their most sensitive components from the commodity OS with only minor source code modifications.

We have built a prototype of our system on top of the Xen Virtual Machine Monitor with Linux as the commodity OS. In practice, we find that the system call routing rules are short and simple – on the order of 10's of lines of code. In addition, applications in Proxos incur only modest performance overhead, with most of the cost resulting from inter-VM context switches.

1 Introduction

While significant effort has been invested into making our computing infrastructure more secure, the number of security incidents continues to increase at an alarming pace. The CERT Coordination Center reports that the number of security incidents increased approximately six-fold in the three years between 2000 and 2003, after which they indicate that incidents had become so commonplace that they were not even worth reporting [4]. Despite these statistics, businesses and individu-

als continue to put increasing trust in computers to store and secure sensitive information, such as financial data, health records, and recently, votes for government elections [15].

Though a great deal of work goes into making operating system kernels more secure, in the vast majority of cases the vulnerabilities being exploited are not in the kernel, but in privileged applications running as user processes. The problem lies not in the reliability of kernel code, but in the overly permissive interface that commodity operating systems (OSs) export, which a privileged application can abuse to make the operating system kernel read or modify the state of any other application. On the other hand, many applications require such privileges to run on a commodity operating system, providing the attacker with many opportunities to take control of the operating system interface. As a result, it seems appropriate that applications that perform security-sensitive operations should have little or no trust in the kernel that lies on the other side of a commodity OS interface.

There have been several attempts to address this situation. One solution is to use a microkernel [1], which minimizes the amount of code running in supervisor mode. However, changing the underlying architecture of the OS kernel without changing the interface that applications use will not give applications any more protection than they currently have. On the other hand, narrowing the application-OS interface requires a large amount of effort to port or rewrite applications currently targeted towards a broad commodity OS interface [23]. There have also been attempts to restrict the interface in existing commodity OSs such as Linux with fine-grained access controls [16]. While effective in principle, the ability to have such controls means that the policy description must be equally fine-grained, making it very complex and time consuming to configure such systems correctly [14]. A third solution is to run the security-sensitive application in its own private OS on a virtual machine (VM) executing on top of a virtual machine monitor (VMM), and thus

^{*}Department of Computer Science, University of Toronto

completely remove all other applications from the trusted computing base (TCB) of the system [10]. This private OS would only support the one application and be specially tailored to its needs. The problem that arises is that applications typically share data and interact with other applications through operating system facilities such as files and pipes. Therefore, short of changing the way applications communicate, we are forced to move the other applications into the private OS as well. As a result, the security-sensitive application is made to tolerate other applications in its TCB that it needs to interact with, but does not necessarily trust.

In this work, we attempt to address these issues by building a system that allows an application developer to choose what operating system facilities should be provided by an untrusted commodity OS, and what facilities need to be provided by a trusted private OS. In this way, applications may continue to use functionality in the commodity OS to communicate with other programs, and avoid having to duplicate functionality in the private OS that does not have to be trusted. This ability is provided by running both commodity and private OSs on a VMM, and using a thin operating system proxy, called *Proxos*, which we have designed. Proxos is a small library that mimics an operating system by handling system calls made by the application.

Proxos takes a novel approach to allowing applications to specify their trust in an operating system. Rather than requiring that the application developer partition the application code into components based on whether they trust the commodity OS or not [23], Proxos only requires the developer to partition the system call interface into system calls that must be trusted and those that need not. Using high-level system call routing rules specified by the application developer, Proxos transparently routes each system call made by the application to the commodity OS if the request does not need to be trusted, or to the private OS if it does. Specifying trust by partitioning the system call interface has the benefit that applications currently implemented for commodity OSs can be easily ported to Proxos with very little effort (typically by only modifying on the order of several hundred lines of code). Consequently, the application developer is able to remove the entire commodity OS from the TCB of their application while maintaining reasonable performance.

In this paper, we make three main contributions. First, we have designed a language that allows developers to configure trust relationships using short and simple system call routing rules. In practice, we find that routing rules can usually be specified in 50 lines or less. Second, we have designed and implemented a prototype of Proxos on top of the Xen VMM [2], with Linux as the commodity OS. We describe the modifications we made to Xen and Linux and evaluate the amount of code that

these modifications add to each component. Finally, we demonstrate the utility of our system by porting three existing applications: a web browser that protects user privacy, an SSH authentication server, and an SSL certificate service used by the Apache web server. We describe the security of the new applications, the issues we encountered in porting them to Proxos, as well as the performance impact of moving to Proxos.

We will start by giving a high-level description of the system architecture needed to run Proxos applications, as well as a description of the Proxos routing language in Section 2. Section 3 follows with an explanation of our prototype, and gives details on modifications we made to Xen and Linux, details on our Proxos implementation, and details on some example private OS functions we have written. To show what applications one might run on Proxos, we describe three representative applications we have ported to Proxos in Section 4, and evaluate the performance impact of Proxos against a vanilla Xen/Linux system in Section 5. Finally, we finish with related work in Section 6, and give our conclusions in Section 7.

2 Overview

In this section, we describe the overall architecture of the system, as well as a description of the security guarantees our system provides. Then, we give a description of the Proxos system call routing language.

2.1 System Architecture

The architecture of our system is illustrated in Figure 1. The system consists of several VMs running on top of a VMM that enforces memory isolation between the VMs and allocates CPU execution time to the VMs. VMs can make hypercalls to the underlying VMM to access resources such as disks and other devices, or to signal or create other VMs. A commodity OS VM runs a commodity OS that provides the facilities usually found in a standard operating system, such as file system implementations, a network stack and a user interface. An administrative VM (not shown in the diagram) contains management tools used to create and manage other VMs. Applications that want to be isolated from the commodity OS are run inside their own private VM along with a Proxos instance. We call such applications private applications. A set of methods inside the private VM implement a private OS, whose purpose is to handle system calls the private application does not trust the commodity OS with.

Proxos handles all system calls made by the application. Depending on the routing rules configured by the

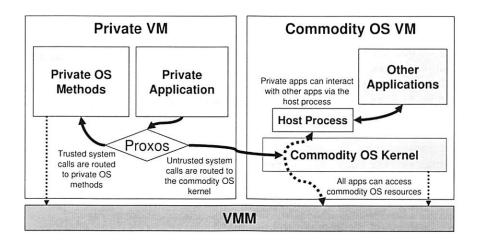


Figure 1: The Proxos System Architecture. Proxos handles all system calls the private application makes by routing them to either the commodity OS or the private OS.

application developer, Proxos will route non-security-sensitive system calls to the commodity OS via inter-VM remote procedure calls (RPCs), and security-sensitive system calls to methods in the private OS. Both Proxos and the private OS are implemented as libraries that are statically linked with the application. As a result, all system calls are converted into subroutine calls to Proxos. The application, along with Proxos and the private OS run on the bare VMM. Since only one application runs in each private VM, all code in a private VM runs in the same protection domain.

To run a commodity application as a private application, the developer first identifies which operating system objects the application uses and that need to be protected from a compromised commodity OS. With this knowledge, the developer identifies the system calls that access these objects and specifies that they are to be forwarded to the private OS using the routing language described in Section 2.3. The private OS methods can be implemented especially for the application by the developer, or even obtained from a library of generic private OS methods provided by a third-party. Section 3.3 describes some private OS methods that we have implemented.

The developer may then have to perform application source code modifications in order to compile it statically, and have it use the facilities that Proxos provides. However, since Proxos exports the same system call interface as the commodity OS, these changes are generally minor. For instance, we were able to port the Glibc library (version 2.3.3) to Proxos with only 218 lines of source code modifications. Next, the private application, the private OS methods, the routing rules and Proxos are all compiled into a single binary, which can be loaded into an empty VM. The developer gives this binary image to the VMM administrator, who registers the new pri-

vate application with the VMM using the administrative VM. Because the private application binaries are stored directly on the VMM, they are safe from tampering by an adversary who has subverted the commodity OS.

To run a private application, a user on the commodity OS invokes a host process, which requests the VMM to instantiate a new VM containing the private application. From this point on, the host process becomes the embodiment of the private application on the commodity OS. The commodity OS attributes any forwarded system call it receives from the private application to the host process that instantiated it. The commodity OS uses the user ID of this host process to make decisions about what operating system objects (such as files or sockets) the application is allowed to access, and also attributes resources used by the forwarded system calls to the host process. In this way, the commodity OS ensures fairness and security between requests made by private applications and requests made by applications running natively on the commodity OS.

Through its host process, a private application can interact with other applications running in the commodity OS by using facilities provided by the commodity OS. For example, by configuring Proxos to forward mknod and open system calls to the commodity OS, a private application can create a named pipe between it and a commodity OS application. Then, by routing read and write system calls to the commodity OS, it can communicate with the commodity OS application by making those system calls on the named pipe. For a communication channel to be created, cooperation is required from both applications, who must agree to communicate, and from the commodity OS, who must agree to fulfill the system call requests made by both applications.

2.2 Security Guarantees

While the commodity OS may at some point become under the complete control of an attacker, we assume that the underlying VMM cannot be subverted and that it continues to enforce isolation between VMs. We also rely on the application developer to properly specify what sensitive components of the interface between the application and the operating system must be protected from the commodity OS. Based on these assumptions, our system maintains the confidentiality and integrity of sensitive private application data even in the face of a compromised commodity OS. The isolation property of the VMM prevents the compromised OS from directly interfering with the private application. The compromised commodity OS can only tamper with system calls that are routed to it by Proxos. However, since these system calls were identified as non-security-critical by the developer, the compromised OS should not be able to affect the private application in any security-critical way. We point out that if the routing rules are specified incorrectly, or if a bug in the application causes it to send sensitive data to an interface that the developer believes should have only held non-sensitive data, then sensitive data could be leaked to the commodity OS. In addition, while the confidentiality and integrity of sensitive private application data are maintained, a compromised OS can impact the availability of a private application by not performing the system calls that are forwarded to it.

So far, we have considered protecting the private application from a potentially malicious OS. However, one could envision the case of a buggy private application that could negatively affect the commodity OS through the system calls it forwards to the OS. However, our design restricts the capabilities of the private application within the commodity OS to that of its host process. Since the private application only has the rights of the user who invoked it, our system does not weaken any existing mechanism that guarantees fairness among users and processes running on the commodity OS.

2.3 The Proxos Routing Language

Proxos may route each invocation of a particular system call differently depending on rules specified by the application developer. For example, Proxos may route read system calls differently depending on what file is being read. We wish to provide a simple and intuitive way for an application developer to partition the system call interface. In principle, one could specify a routing rule for each of the over 200 system calls that a commodity OS like Linux provides, but this would be complex and time consuming. Further, we do not believe it necessary in most cases to have such fine-grained control over

system call routing. We organize system calls by the resources they access and create a Proxos routing language with which the developer can specify routes for those resources. In this language, the operating system provides six resource classes to an application: persistent storage (disk), user interface, network, randomness, system time, and memory. Peripheral devices such as printers, USB devices, etc, are abstracted by the OS into file objects and are thus part of the persistent storage category.

While it is possible to provide routing rules for all six resources, we have found that this is unnecessary. An application may choose to forward system requests to the commodity OS for two reasons: either it wants to use the resource as a communication channel with another application, or it does not need the resource to be trusted and thus wishes to include the resource outside of its TCB. As a result, persistent storage, user interface and the network are routed by Proxos because these are resources that applications either use to communicate, or may not need to trust. System time and randomness are never routed because they cannot be used as communication channels, and are provided by the underlying VMM without increasing the application's TCB. Finally, memory related system calls (such as brk and mprotect) are used to indirectly manipulate page table entries. However, a private application would never trust a commodity OS with control of its page tables since this would imply granting the commodity OS access to the private application's memory. Therefore, it does not make sense to route memory-related system calls. All non-routable system calls are directed to functions provided by Proxos.

Based on this model of operating system resources, we have designed a simple language that allows the application developer to specify which system calls will be routed to the commodity OS, and which to the private OS. Figure 2 shows a stripped-down example of a routing specification in our language. Lines prefixed with a "#" are comments. The Rules section consists of three declarations, one for each of the routable resource classes. The specifications for the disk and network resource classes are a list of tuples, where each tuple describes the particular resource, and a table of function pointers used to access the resource. In this case, the specification for the user interface (UI) has "*" as a resource description because the application wants to route all three standard I/O streams (i.e. stdin, stdout, and stderr) to the private OS. The example also specifies that access to any file with name /etc/secrets should be handled by methods in the private OS. The same is true for system calls to any UNIX domain socket bound to tmp/socket and to any TCP socket with a peer IP address of 192.100.0.4 on port 1337. By default, Proxos will route all system calls to resources that do not match

```
# Rules Section
# route accesses to /etc/secrets to private OS
DISK: ("/etc/secrets", priv_fs)
# route accesses to UNIX domain socket bound
# to /tmp/socket and TCP socket bound to peer
# 192.100.0.4 port 1337 to private OS
NETWORK:("unix:/tmp/socket", priv_unix),
        ("tcp:192.100.0.4:1337", priv_tcp)
# route all accesses to stdin, stdout
# and stderr to private OS
UI: (*,priv_ui)
# Methods Section
 individual methods in the private OS
# that are bound to system calls
priv_fs = {
  .open = priv_open,
  .close = priv_close,
  .read = priv_read,
  .write = priv_write,
  .lseek = priv_lseek
```

Figure 2: Routing Example. This example shows a simple set of routing rules that protects operations on a particular file, two network sockets, and the standard I/O streams.

any rule to the commodity OS.

The Methods section defines which methods in the private OS will handle system calls from the application. When the application attempts to open the file /etc/secrets, Proxos will call the priv_open method in the private OS to handle the request and return a file descriptor. All subsequent system call operations (such as close, read, write and lseek) on the file descriptor associated with that file will also be forwarded to the associated private OS method in the table. On the other hand, any system call on the file that is not in priv_fs will be forwarded to the commodity OS. Method tables for priv_ui, priv_unix and priv_tcp are not shown in the figure, but must also be specified by the application developer.

Rather than specifying trust policies by partitioning code, or by restricting abilities, specifying policies by partitioning interfaces to resources results in a more compact and intuitive policy description. Further, our specification language allows the application developer to use the same names for resources as those in the source code, making the routing rules easier to write and understand.

3 Prototype Implementation

There were several requirements that dictated which underlying system we chose to implement our Proxos prototype on. First, we needed a way of "hoisting" a commodity OS to a lower privilege level and inserting our own privileged code beneath it. Second, the system had to provide isolation between the private applications and the commodity OS, but at the same time allow some controlled communication between them. In light of these requirements, we eventually settled on using the Xen VMM [2] and Linux as our commodity OS for our experimental substrate. However, we believe that the features required by our system could be provided by any VMM or microkernel.

In this section, we describe the three main components we implemented in building our prototype. First, we describe our modifications to Xen and Linux to provide support for starting private applications, and to forward system calls between VMs. Second, we describe our Proxos operating system proxy prototype, which routes system calls to either the commodity Linux kernel or to private OS methods. Finally, we describe some private OS methods that we have implemented.

3.1 Modifications to the VMM and the Commodity OS

Modifications made to Xen and the Linux kernel can be categorized into three components: the start-up and shutdown of private applications, a facility for forwarding system calls between VMs, and a trusted path facility.

Since the Linux kernel and private applications do not trust each other, the private application start-up process must make several guarantees. First, the private application must not be able to gain any privileges beyond those of its host process. This implies that the Linux kernel must always be able to attribute system calls routed to it to the host process that initiated the private application forwarding the system call. Second, a compromised commodity OS should not be able to initialize a private application in an unsafe state. Finally, the private application should not be able to access any Linux kernel memory that the kernel has not authorized it to.

The VMM administrator registers private applications with the VMM via a configuration file. This file assigns a name to each private application and sets start-up parameters for each private VM. Later, when the host process starts a private application, it will use this name to indicate to the VMM which private application to start. Figure 3 describes the private application start-up process used in our prototype in detail.

In Step 1, private applications are started using the pr_execve system call that we added to the Linux kernel. pr_execve is the private application analog to the execve system call and, like execve, takes the name of the private application to be started as its argument. pr_execve causes the current process to become the host process for the private application.

In Step 2, the Linux kernel allocates a shared buffer that is used later for system call arguments forwarded to

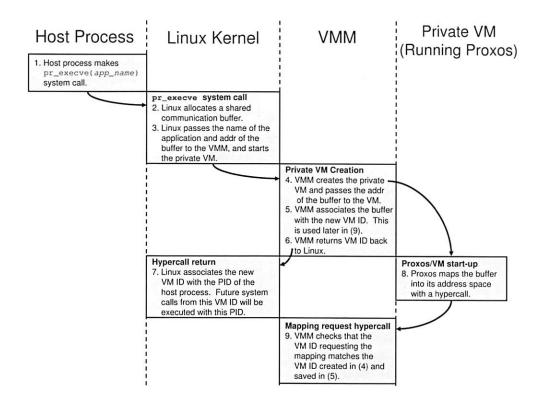


Figure 3: Private Application Start-up Sequence. The steps are arranged into columns with the titles at the top indicating what system component each step takes place in.

it from the private application. The kernel passes the address of this buffer to the VMM in Step 3, and at the same time signals the administrative VM to start a new VM for the private application with a hypercall we introduced. The administrative VM will only start the private VM with parameters set by the system administrator, ensuring that even a compromised Linux OS can only start private applications from a known, safe state. Note that a compromised Linux OS may start a private application different from the one the host process requested and attempt to get the user to use the wrong private application. To detect this, Proxos relies on application level safeguards such as the trusted path used in our web browser or cryptographic keys used in our SSH private server. We will discuss both of these applications in Section 4.

In Steps 4 to 7, the VMM creates a new private VM for the application, informs the Proxos in the private VM of the location of the shared communication buffer, and informs the Linux OS of the identity of the new VM (by giving it a VM ID). Then, in Step 8, Proxos tries to map the shared buffer into its address space via another hypercall. Originally a privileged hypercall, we modified this hypercall so that private VMs may use it. However, we also added an extra check to ensure that the VM making the mapping request in Step 9 is the same as the one to which the VMM originally passed the shared buffer

address in Step 4.

Private application shutdown is much simpler as there are no security guarantees to be made. If the private application initiates the shutdown, then it informs the VMM via a standard hypercall. We extended this hypercall to notify the Linux kernel, so that the kernel may terminate the host process accordingly. Even if the private application has not terminated, the kernel may still forcibly destroy the host process (by killing the process). However, the kernel does not have the privileges to force the private application to shutdown, so by killing the host process, the kernel can only revoke the private application's ability to access commodity OS resources.

Another set of modifications allow private applications to forward system calls to the Linux OS. The goal is to reduce the latency of forwarded system calls by reducing the number of domain crossings. Xen already provides a facility that allows VMs to send events to each other. By combining this with the shared buffer between the Linux OS and the private application, we were able to add a simple RPC mechanism to Xen. We then made modifications to the Linux kernel to allow it to efficiently execute forwarded system calls. When the Linux kernel receives the system call arguments, it determines the appropriate host process to wake up by examining the source of the RPC and comparing that to information it recorded in

Step 7 of the start-up sequence. As the host process is about to be scheduled, a trip into user-space can be saved by placing the system call arguments in the appropriate registers and transferring control directly to the system call handler in the kernel. When the system call handler completes, the kernel sends the return value back to the private application via an RPC response message and returns the host process to the queue it was in before the forwarded system call arrived. As a result, our prototype handles forwarded system calls without any domain crossings in the Linux OS.

Finally, we also needed the VMM to provide a trusted path facility so that private applications can communicate directly with the user without having to trust the commodity Linux OS. This would prevent a compromised Linux OS from masquerading as a private application, as well as prevent a compromised Linux OS from eavesdropping on communication between a user and a private application. To support this, the VMM provides user interface facilities such as a console driver and graphical window system. If the private application wants to use these facilities, it routes system calls on standard I/O streams (i.e. stdin, stdout and stderr) to private OS methods, which will forward the requests to the VMM console driver. Similarly, it routes X window operations to private OS methods that will translate them into the appropriate operations on the VMM window system. The implementation of minimal trusted window systems on secure kernelized systems has been studied in the literature [9, 22]. Rather than reimplement these in our prototype, we simply provided an emulation of their functionality, but do not make any effort to reduce the amount of code that is added to the VMM. We did this by running an X server on Xen's administrative VM and using nested X servers to give each VM its own separate X interface.

We found that modifying Xen and Linux to allow private application start-up and shutdown, as well as forwarded system calls, had very little impact on the size of the Xen TCB. Many of the facilities needed were already present in the Xen VMM and we only had to make these accessible to unprivileged VMs and add checks to make sure they could not be abused. The only component that increases the code base of the VMM significantly is the graphical user interface. A significant portion of this component can be implemented outside of the trusted computing base of the VMM [9, 22], but exploring the design of trusted window systems was not a goal of our prototype.

3.2 The Proxos Prototype

Our prototype is derived from the *Minimal OS* example that comes with the Xen 2.0 source code. Proxos runs

in a single address space and supports only one private application. Our current implementation is also single-threaded, although we plan to support threads in the future. Apart from providing basic memory and page table management, Proxos also contains: a block driver that supports raw accesses to a private block device exported by the VMM; and a console driver that provides direct access to the Xen console. Our prototype does not provide a TCP/IP stack or a network driver. We found these unnecessary as many security-sensitive applications already assume the network is not trustworthy and employ cryptographic safeguards such as SSL to protect network communications. This allows us to safely reuse the network services of the commodity OS.

Proxos uses operating system abstractions to determine where to route system calls at run time. In the case of Linux, the abstraction used by applications to access resources is a file descriptor. Initially a file descriptor is bound to a resource via a system call such as open or socket. Subsequent operations on that resource are then performed by naming the descriptor in the system call.

The design of Proxos is very simple, and is similar to the way virtual file system methods are implemented in Linux. Routing rules for the application are converted into lookup tables, which are then compiled into the Proxos library and linked with the private application. When descriptors are created, Proxos compares the name of the resource they are being bound to with the routing rules specified for the application. For example, if a file descriptor is being created via an open system call to a file, Proxos compares the name of the file being opened with the list of tuples provided in the DISK resource class. If a match is found, Proxos uses methods from the method table specified in the matching routing rule to handle subsequent system calls on the descriptor. Proxos provides a set of default methods which route untrusted system calls to the Linux OS. If a routing rule specifies a private OS method to be called, Proxos transfers control to the appropriate location in the private OS.

The private application uses file descriptors to name objects in both the private OS and the commodity OS. File descriptors in the commodity OS are allocated from a name space independent of the one the private application is using. Upon opening a new file in the commodity OS, Proxos may find that the commodity OS has assigned a file descriptor number that the private application is already using to name another object in the private OS. As a result, Proxos translates between the file descriptors used by the private application, and those used in the commodity and private OSs.

Most routable system calls can be routed transparently to the Linux OS. However, the fork, execve and select system calls have slightly different semantics.

When forwarded, the fork system call will cause the host process in the Linux OS to fork. The forwarded fork creates concurrency on the Linux OS side, but the application in the private VM will still contain only a single thread of execution, so parent and child code must be executed sequentially. After the fork, the private application specifies whether system calls it forwards to the Linux OS should be executed by the parent process or the child process. This is done by setting the target PID flag in Proxos to indicate the process ID (PID) of the process that should be the recipient of system calls forwarded to the Linux OS. The value of this flag accompanies every system call Proxos forwards to the Linux OS. The Linux OS checks that the PID specified by the flag belongs to either the host process, or a child of the host process. These semantics imply that forwarding fork system calls requires the developer to make any concurrent code sequential in the private application. To support standard fork semantics, the underlying VMM needs to be capable of duplicating the address space of the private application (preferably using copy-on-write for efficiency). While we did not support this in our prototype, we note that others have proposed adding such functionality to Xen [24].

The semantics of forwarded execve system calls are also slightly different. If the execve system call is made without a fork, the host process will terminate and a new program will take its place. If the new process is not willing to host system calls forwarded to it, the private application will be unable to forward system calls to the Linux OS. More commonly, a recently forked process will execute execve. In this case, the private application will lose the ability to forward system calls to the child, but retain the parent as the host process. More details on how fork and execve are used in private applications will be given in the description of our port of the SSH server in Section 4.2.

Finally, select has a slightly different behavior under Proxos than its Linux counterpart. select allows applications to listen on several file descriptors simultaneously and notifies them when there is activity on any of the descriptors. In Proxos, an application may execute a single select on file descriptors from both the commodity OS and the private OS. However, Proxos forwards system calls by making synchronous inter-VM RPCs. This limitation of our current prototype prevents Proxos from routing select system calls to both OSs simultaneously, so it serializes them and imposes a timeout on each select call. Proxos will alternate between which OS to execute select on first to ensure no file descriptor is starved. The poll system call has the same behavior as select in our system. The consequence of this is that events on file descriptors that happen close together may not be delivered to the private application in the same order that they occurred because Proxos may be polling the other OS instance when the first event occurs. However, we have not seen this to be an issue and, to the best of our knowledge, Linux makes no such ordering guarantees either.

3.3 Private OS Methods

In our prototype, we have implemented two example private OS components: one that implements a private file system, and one that implements a trusted path by forwarding standard I/O streams and X window messages to the VMM.

A private file system allows the private application access to persistent storage that is protected from tampering by the Linux OS. We wanted to implement this by adding as little code to the private VM as possible, as any code we add increases the TCB of the application. Rather than implement an entire file system, our private file system outsources most of its functionality to the commodity Linux OS through forwarded system calls, but maintains the secrecy of any information stored by encrypting all data before writing it to the Linux file system [12]. To protect the data from tampering and replay, hashes of all files stored on Linux by the private file system are kept on a private block device available directly from the underlying VMM. Doing this significantly simplifies the file system implementation, as all that is needed are the cryptographic functions, some code to manage file system buffers, and block device drivers to store the file system hashes. The drawback is that a compromised Linux OS could potentially deny the private application access to files that the private file system has saved. However, our applications typically depend on the Linux OS for other services as well, so no forward progress guarantees are broken by this.

In our prototype, the private OS implements a trusted path by routing operations on standard I/O streams and the X server's socket to the VMM. The private OS methods translate system calls on standard I/O streams to operations on Xen's console driver and route system calls on the X server to the administrative VM. A host process on the administrative VM then executes the routed system calls on a socket connected to a nested X server instance that is separate from the one that the commodity Linux OS is using.

3.4 Discussion

With the exception of modifications to the Linux kernel, all components implemented in our prototype will be part of the application TCB. As a result, we placed a lot of emphasis on keeping the impact on code size and complexity small, especially with respect to the VMM. One

Component	Lines of Code	
VMM modifications	656	
Linux modifications	4380	
Proxos	7348	
Private File System	1817	
Trusted Path	1313	

Table 1: Number of lines of code in each component in our Proxos prototype. The VMM modifications do not include the X server running in the administrative VM.

caveat is that Proxos does not need to support every system call that Linux exports. For example, Proxos does not support system administration calls, such as those to control swap devices, or load and unload kernel modules, as private applications will not need to make such requests. Out of the 289 system calls of the Linux 2.6.10 kernel, our Proxos prototype only needs to support (either internally or by forwarding) 56 of them to run most applications. However, we fully expect this proportion to increase as Proxos matures. The size of the components in our prototype are given in Table 1.

4 Applications

In this section, we describe three applications that we have ported to Proxos. We selected applications that will benefit from partially trusting a commodity OS, and illustrate interesting issues that arose when porting them. Our first application is a secure web browser that protects user information. Our second application is an SSH server that protects system-critical information such as passwords and host keys even if the commodity OS is compromised, but still allows users who login to gain a full shell on the commodity Linux OS. Our final application is an SSL certificate service that we use with an Apache web server to implement SSL transactions. In this case, the private keys corresponding to the certificate are protected.

4.1 Secure Web Browser

A serious threat to the security and privacy of users is spyware, which is malicious software that is surreptitiously installed on machines and monitors the web surfing habits of users. While the goal of most spyware is to collect usage data for marketing, spyware has been shown to decrease the security of user system by recording and transmitting confidential information that it has collected [18, 20].

Spyware collects information by either monitoring the user's keystrokes, or by scraping files where web browsers have recorded user information. We ported Dillo [5], a simple graphical web browser, to Proxos and configured the routing rules so that all user I/O is directed to the trusted VMM user interface, thus creating a trusted path, and all sensitive data that Dillo reads from disk or writes to disk is directed to our private file system. No other rules are specified, and thus other network operations such as HTTP requests are routed to the Linux OS by default (for extra security, the user should use HTTPS to encrypt traffic between the browser and the web server to prevent any spyware on the Linux OS from observing or tampering with it). Similarly, any documents or executables that the user downloads from the Internet are saved to the Linux file system. In addition, any external helper applications that Dillo invokes will be transparently created and executed on the Linux OS.

For the most part, no source code modifications were required to port Dillo. The only necessary modifications were due to Dillo's use of graphical themes, which are implemented as code that is dynamically loaded at run time based on the theme the user selects. In our prototype, it is not safe to load code from the Linux OS, since an adversary may have tampered with it. To support the default theme, we removed the code that loads themes at run time and statically linked the default theme into the Dillo private application. In theory, code could be safely loaded from the Linux OS if encrypted and accompanied by a valid signature that the private application could verify, but our current prototype does not support this.

4.2 SSH Authentication Server

Often when attackers compromise a system, the system administrator is not only forced to rebuild the entire system from scratch to ensure that any malicious software has been removed, but also to perform the arduous task of tracking down every user and ensuring that they change their passwords in case the attacker has been able to learn some of the old passwords. Similarly, she must change any cryptographic host keys, which the machine uses to authenticate itself, and distribute new keys to all parties that the machine interacts with. Being able to ensure the secrecy of user passwords and the host keys of a system after a security compromise would save the administrator significant time and effort.

To demonstrate the utility of Proxos in protecting the secrecy of sensitive data, we ported the OpenSSH authentication server (version 3.9p1) to Proxos. The OpenSSH server accesses several sensitive resources including configuration files, the password file and the host key file. We wrote routing rules to store the password file, host key, and global configuration files on the private file system. The SSH server also performs network operations, but no rules are specified for NETWORK resources since OpenSSH is designed to function with an untrusted

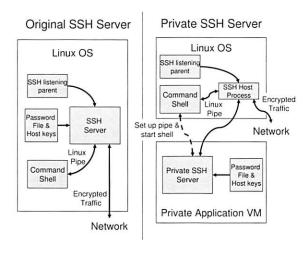


Figure 4: Comparison of the original SSH server and our private SSH server.

network. Other than the routing rules, only two modifications involving the fork system call were required to implement a private SSH server application. The architecture of our new private SSH server is shown alongside the architecture of the original SSH server in Figure 4. One modification arose because the SSH server requires some concurrency to allow multiple users to authenticate simultaneously. The native version of SSH handles this by having a parent process listen on the SSH port, and then spawning a child for every connection the parent receives. Our private SSH server still has the listening parent as a native Linux application, but implements the children as private applications. When the listening parent detects a new connection, it forks a child (on the Linux OS), which then uses prexecve to instantiate a private SSH server VM, and in doing so becomes the host process for the new VM.

The private SSH server starts-up and reads the sensitive data from the private file system, and then proceeds with user authentication. If a user logs in using private key authentication, the private SSH server will need to access the public keys the user has placed in a file in their home directory on the Linux OS. Proxos provides access to the user's keys without any extra configuration - since the user's key files do not match any routing rules, requests to them will be forwarded to the Linux OS by default. If the user authenticates successfully, the native SSH server forks a child that will execute a command shell. Before the child starts the command shell, the native SSH server creates a pipe between itself and the command shell redirecting all input and output from the shell to itself, so that it can encrypt any shell output before sending it to the network, and decrypt any shell input coming from the network. In our version, the private SSH server changes the Proxos target PID flag to point to the new child after the fork, and then executes the child code, forwarding the system calls required to set up the pipe and start the command shell. After this, it changes the target PID flag back to the parent and executes the SSH server code. The shell will pipe all input and output through the host process to the private SSH server, which encrypts and decrypts data as appropriate between the shell and the network.

4.3 SSL Certificate Service and Apache

Next, we explored the performance impact of Proxos on Apache with SSL. As in the SSH server, the Apache server relies on concurrency so we only ported the crypto library portion of the OpenSSL library to Proxos, and left the Apache web server on the Linux OS. The crypto library uses confidential private keys stored in the SSL certificate, which would be protected if the web server was compromised. Our port uses Apache version 2.0.52 and version 0.9.7g of the OpenSSL library.

To setup SSL sessions, Apache makes calls to the OpenSSL library, which uses the OpenSSL "engine" interface to invoke the crypto library. We modified the engine interface to spawn a private application that will use the private key of the server's SSL certificate to sign challenges during an SSL handshake. Unfortunately, this operation is called on every HTTP request that uses SSL (i.e. an HTTPS request), and would give very poor performance because each request results in the instantiation and shutdown of a private VM. To remove the frequent instantiation and shutdown of the private VM, we modified Apache to spawn a process when it starts-up, which will act as the host process for a single private SSL certificate application. Apache was also modified so that a portion of the shared buffer between the host process and the private application is mapped into the address space of each Apache thread. To process an HTTPS transaction, a thread enqueues the signing request on the shared buffer, sends a signal to Proxos for processing and sleeps until the request has been processed. Since multiple Apache threads will be accessing the shared buffer, we also added the appropriate synchronization between the threads to prevent races.

4.4 Discussion

The size of our routing rule descriptions, along with the lines of code that were modified for each of the applications, as well as Glibc (version 2.3.3), is given in Table 2. In porting these applications we found that what often takes some time are modifications to application source code that are required to support operations like fork and execve in the private SSH server, or to statically

Application	Rules	LOC Modified
Dillo	53	22
SSH Server	35	108
Apache & OpenSSL	28	667
Glibc		218

Table 2: Size of routing rules and number of LOC modified for each application.

link in dynamic code in Dillo. Apache required more effort since several threads could make challenge-signing requests simultaneously, and this required careful arbitration and synchronization to preserve performance. We find these results encouraging – Proxos enables the application developer to remove the entire commodity OS kernel and privileged applications from the TCB of the private application by modifying on the order of several hundred lines of code in the application, and writing around 50 lines of routing rules.

5 Performance Evaluation

The performance of VMMs versus native operating systems has been well studied in the literature [2, 3]. To ascertain the overhead introduced by Proxos, we compare the performance of our system against a system running an unmodified Linux kernel executing on an unmodified Xen VMM. We first use microbenchmarks to better understand the components that contribute to the cost of making forwarded system calls from a private application. Then we evaluate the performance of the SSH and Apache/SSL Certificate applications described in Section 4 on our system. All tests were performed on a machine with a 3GHz Intel Pentium 4 processor, 1GB of RAM, a 7200 RPM Serial-ATA disk with 8.9 ms seek time, and a 100Mb Ethernet NIC. Our prototype is built on Xen 2.0, with Fedora Core 3 Linux running a 2.6.10 kernel as the commodity OS, and its performance is compared against vanilla versions of the same software. For our runs, 768MB of RAM were allocated to the commodity OS, and the rest was used for Xen, the administrative VM, and private applications.

5.1 Microbenchmarks

To analyze the overhead of a system call forwarded from the private application to the Linux OS, we must first understand the individual components that make up a forwarded system call. These costs are illustrated in Figure 5. In Step 1, Proxos sends an event to the Linux kernel, notifying it of the forwarded system call, and then yields the processor, causing a VMM context switch into the Linux OS VM. In the Linux kernel, a virtual interrupt

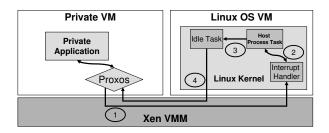


Figure 5: Breakdown of costs incurred in a forwarded system call.

Benchmark	Linux	Proxos	Overhead
NULL system call	0.37	12.88	12.51
fstat	0.57	14.28	13.71
stat	8.76	25.98	17.22
open & close	14.57	47.18	32.61
read	0.45	13.51	13.06
write	0.42	13.24	12.82

Table 3: Forwarded system call latencies on LMbench microbenchmarks. All measurements are given in μ s.

handler receives the event and enqueues the system call request on the process descriptor of the host process. In Step 2, we wait until the host process is scheduled. On a lightly loaded system, this incurs only the cost of another context switch within the Linux kernel, but may take more time if the Linux kernel is heavily loaded. After the Linux kernel executes the system call, it will not yield the processor back to the VMM until either the VMM scheduler decides to preempt the Linux OS VM, or the kernel runs out of runnable processes and schedules the idle task in Step 3. Finally, in Step 4, another VM context switch occurs and Proxos can receive the result of the system call. While this accounts for four context switches, there is actually a fifth context switch because Xen will schedule the administrative VM in either Step 1 or Step 4.

We ran the system call latency benchmarks in the LM-bench 2.5 microbenchmark suite [17] in a private VM configured to forward all system calls to an idle Linux OS VM, and summarize our results in Table 3. We also used the context switch microbenchmark in LMbench and measured the minimum cost of a context switch to be $2.88\mu s$ on our machine. As a result, the expected five context switches would take approximately $14\mu s$, which tracks well with the measured results. This cost is added to every system call except for stat and open, whose larger overhead can be explained by the fact that each context switch changes virtual to physical page mappings, and causes a TLB flush. Since both stat and open take a filename as an argument, the Linux kernel

must make several queries to the buffer cache to find the correct inode (LMbench ensures that the inodes required to access the files are cached in memory), which will result in TLB misses. These misses do not occur when the benchmarks are run directly on Linux because the kernel never switches to another process, so no context switches occur.

5.2 Application Benchmarks

We now evaluate the overhead imposed on our private SSH server and SSL certificate service. Like our microbenchmarks, applications incur overhead when system calls are routed to the commodity OS. To evaluate the average overhead a forwarded system call experiences, we used an SSH client to login to our private SSH server over the loopback device and measured the time taken to copy files ranging from 32MB to 256MB over the SSH connection. Each file transfer was performed five times on both the private SSH server and a native SSH server running on Xen. The standard deviation was less than 1.5% across our measurements. Figure 6 plots the average difference in time taken by the private SSH server over the native SSH server to transfer a file, against the number of forwarded system calls the private SSH server made. We perform linear regression on the average values and found a correlation of 0.92, indicating that the overhead is well correlated with the number of system calls. We then estimate the start-up component to be 0.72s and the per-system call cost to be 15.7μ s. For large files, where the cost of start-up has been amortized, the private SSH server only takes 6.0% longer to transfer the same file as the native SSH server. Note that since this overhead is comparable to the variance in our measurements, the estimated system call overhead should not be taken too literally, and is merely a rough approximation.

We suspected that a large part of the start-up cost for the private SSH server is due to VM creation. We confirmed this by measuring the time to start an empty private VM, which is approximately 0.35s. Starting a Xen VM requires the use of several user-space scripts in the administrative VM, making it very expensive, and we have not made any effort to optimize this operation. The remaining 0.37s is the time the private SSH server uses for initialization, which includes the time it takes to read in sensitive data from the private file system. This operation requires several cryptographic operations to decrypt the data and verify the authenticity of files stored on the commodity OS file system.

To evaluate the performance impact of our private SSL certificate application, we used Mindcraft's Webstone benchmark [25] extended with SSL. We configured the benchmark with 150 SSL clients, which was enough to

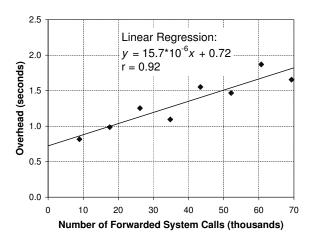


Figure 6: SSH benchmark. We plot the overhead of the private SSH server versus the number of system calls forwarded to Linux.

fully load an Apache server on Xen. The same number of clients was used to measure the amount of bandwidth our Proxos-enabled web server could support. We expected the Proxos-enabled web server to introduce low overhead because HTTPS transactions mainly perform computation and make very few system calls. Our experiments show that there is actually a slight increase in throughput – 5.04Mb/s for the Proxos-enabled web server as compared to the native web server's throughput of 4.75Mb/s. From this, we surmise that the overhead Proxos introduces is very low and that the changes we made porting the system likely perturbed the system in such a way as to produce a slight performance gain.

6 Related Work

While isolating processes from operating system components bears some similarities to multi-server microkernels [1, 11], Proxos is more similar in structure to Exokernels [8] and the Denali kernel [26], with their statically linked, single user LibOSs. Where Proxos differs from these systems is in its objective – while the goal of the former systems is to give applications some ability to customize OS management of their resources, Proxos gives applications the ability to customize the trust relationship between applications and the OS kernel.

Various systems aim to limit the damage a compromised application can cause. SELinux [16] allows the administrator to set a fine-grained mandatory access control policy for the system, thus limiting the privileges an attacker would gain by hijacking an application. However, fine-grained control has its costs. SELinux policies are large and complex – the size of the default pol-

icy set for the Fedora Core 3 Linux distribution has over 290,000 rules and consumes more than 7MB of kernel memory. In contrast, our interface routing configurations are typically around 50 lines long or less. In addition, because SELinux works by restricting the abilities of applications, its policy rules must define all the permitted behaviors of every application on the system. Since Proxos operates by isolating security-sensitive applications from the rest of the OS, Proxos policy rules only need to be defined for the applications being protected. Asbestos [6], Eros [21] and Singularity [13] also limit information flow and privileges, but through mechanisms significantly different from Proxos. Asbestos uses process labels that are updated dynamically, combining aspects of capabilities and information flow policies. Eros is a pure capabilitybased microkernel and Singularity only permits communication between processes through strongly typed and formally verified channels. All of these paradigms require applications to be ported to fundamentally different application interfaces. By keeping the same application interface as a commodity OS, Proxos does not require any extensive porting for existing applications. Further, applications that do not require Proxos can remain in the commodity OS and suffer no overhead.

Terra [10], Nizza [23] and Microsoft's NGSCB [7] are projects that propose new operating system models to increase the security of applications. Terra provides coarse-grained isolation by enclosing security-sensitive applications along with their own operating system in a "closed-box" VM. Applications may only communicate with applications on other VMs via the network interface. Similarly, NGSCB runs specialized "agents" in a high-assurance OS called the Nexus, which is isolated from a standard Windows OS by a VMM. In both NGSCB and Terra, each OS must contain all the functionality required by the application, even if the functionality does not have to be trusted by the application, while Proxos can reuse untrusted functionality in the commodity OS. On the other hand, Nizza, along with projects μ -Sina [12] and Perseus [19], take a fine-grained approach to minimize the amount of code in an application's TCB. They propose heavily modifying the application source code to extract and port the security-sensitive components of an application to a microkernel. The resulting applications often have reduced functionality. In contrast, by partitioning trust along a commodity OS interface, Proxos allows applications to retain the ability to communicate with untrusted applications through standard OS facilities, which is lost in Terra. At the same time, it avoids the effort to modify application source code or the reduced application functionality that Nizza

Finally, nothing in this work is Xen-specific. While many features of Xen made the prototype easier to build,

we believe that the Proxos infrastructure is applicable to other hypervisor based VMM systems, such as VMware ESX Server and Microsoft's forthcoming Viridian.

7 Conclusions

Current commodity OSs export an interface that is too permissive to privileged applications, allowing compromised applications to gain control of the operating system kernel and attack other applications. Proxos allows applications to partition the interface between them and the commodity OS kernel into trusted and untrusted components by specifying system call routing rules. The end result is that Proxos allows application developers to protect applications from a compromised kernel without having to make major source code modifications.

By building a Proxos prototype and porting several representative applications, we have found that specifying trust at the system call interface is a powerful and simple way of isolating applications from the operating system. Proxos routing rule specifications are short and simple, and can be expressed in 10's of lines of code. Minor source code modifications are also required to support applications, mainly due to the semantics of the fork system call, and to remove any instances of dynamically loaded code that cannot be eliminated by static linking. In cases such as our web server, where expensive VM start-up and shutdown may become very frequent, further modifications are necessary to preserve performance. We expect that in most cases, a single graduate student who is familiar with an application can port it to Proxos in a day or two. With a modest cost in engineering time and a reasonable impact on application performance, system call routing enables the developer to protect the secrecy and integrity of applications from a compromised operating system.

Acknowledgements

We are grateful to Nagendra Modadugu, who generously provided us code for his SSL-enabled version of Webstone. Chandu Thekkath provided insightful discussions that helped initially in this work. Various iterations of this paper were improved immensely by comments from Tom Hart, Ian Sin, Jesse Pool, Michael Stumm, Ashvin Goel, Reza Azimi, Troy Ronda, and others in the SSRG group at the University of Toronto. We would also like to thank Derek McAuley, our shepherd, for his helpful suggestions. This work was supported in part by an NSERC Discovery Grant and a MITACS seed grant.

References

- [1] M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of Summer Usenix*, July 1986.
- [2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, Oct. 2003.
- [3] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997)*, pages 143–156, Oct. 1997.
- [4] CERT Coordination Center, 2006. http://www.cert.org.
- [5] Dillo web browser, 2006. http://www.dillo.org.
- [6] P. Efstathopoulos, M. Krohn, S. Van De Bogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005), pages 17–30, Oct. 2005.
- [7] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Will-man. A trusted open platform. *Computer*, pages 55–62, July 2003
- [8] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 251–266, Dec. 1995.
- [9] N. Feske and C. Helmuth. A Nitpicker's guide to a minimalcomplexity secure GUI. In *Proceedings of the 21st Annual Com*puter Security Applications Conference (ACSAC 2005), pages 85–94, Dec. 2005.
- [10] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 193–206, Oct. 2003.
- [11] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings* of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997), pages 66–77, Oct. 1997.
- [12] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB size by using untrusted components small kernels versus virtual-machine monitors. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Sept. 2004.
- [13] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Microsoft Corporation, Redmond, WA, Oct. 2005.
- [14] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, pages 59–74, Aug. 2003.
- [15] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach. Analysis of an electronic voting system. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 27–42, May 2004.
- [16] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In FREENIX Track of the 2001 USENIX Annual Technical Conference (FREENIX'01), pages 29–42, June 2001.

- [17] L. W. McVoy and C. Staelin. LMbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Usenix Technical Conference*, pages 279–294, Jan. 1996.
- [18] A. Moshchuk, T. Bragin, S. D. Gribble, and H. Levy. A crawler-based study of spyware in the web. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS 2006)*, Feb. 2006.
- [19] B. Pfitzmann, J. Riordan, C. Stueble, M. Waidner, and A. Weber. The PERSEUS system architecture. Technical Report RZ 3335, IBM Research Division, Sept. 2001.
- [20] S. Saroiu, S. D. Gribble, and H. M. Levy. Measurement and analysis of spyware in a university environment. In 1st Symposium on Networked Systems Design and Implementation (NSDI 2004), pages 141–153, Mar. 2004.
- [21] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP 1999)*, pages 170–185, Dec. 1999
- [22] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *Proceedings of the 13th USENIX Security Symposium*, pages 165–178, Aug. 2004.
- [23] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of EuroSys* 2006, Apr. 2006.
- [24] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 148–162, Oct. 2005.
- [25] Webstone: The Benchmark for Web Servers, 2006. http://www.mindcraft.com/benchmarks/webstone/.
- [26] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Sympo*sium on Operating Systems Design and Implementation (OSDI 2002), pages 195–209, Dec. 2002.

Connection Handoff Policies for TCP Offload Network Interfaces

Hyong-youb Kim and Scott Rixner

Rice University

Houston, TX 77005

{hykim, rixner}@rice.edu

Abstract

This paper presents three policies for effectively utilizing TCP offload network interfaces that support connection handoff. These policies allow connection handoff to reduce the computation and memory bandwidth requirements for packet processing on the host processor without causing the resource constraints on the network interface to limit overall system performance. First, prioritizing packet processing on the network interface ensures that its TCP processing does not harm performance of the connections on the host operating system. Second, dynamically adapting the number of connections on the network interface to the current load avoids overloading the network interface. Third, the operating system can predict connection lifetimes to select long-lived connections for handoff to better utilize the network interface. The use of the first two policies improves web server throughput by 12-31% over the baseline throughput achieved without offload. The third policy helps improve performance when the network interface can only handle a small number of connections at a time. Furthermore, by using a faster offload processor, offloading can improve server throughput by 33-72%.

1 Introduction

In order to address the increasing bandwidth demands of modern networked computer systems, there has been significant interest in offloading TCP processing from the host operating system to the network interface card (NIC). TCP offloading can potentially reduce the number of host processor cycles spent on networking tasks, reduce the amount of local I/O interconnect traffic, and improve overall network throughput [14, 18, 27, 30, 31]. However, the maximum packet rate and the maximum number of connections supported by an offloading NIC are likely to be less than those of a modern microprocessor due to resource constraints on the network inter-

face [24, 28, 30]. Since the overall capabilities of an offloading NIC may lag behind modern microprocessors, the system should not delegate all TCP processing to the NIC. However, a custom processor with fast local memory on a TCP offloading NIC should still be able to process packets efficiently, so the operating system should treat such a NIC as an acceleration coprocessor and use as much resources on the NIC as possible in order to speed up a portion of network processing.

Connection handoff has been proposed as a mechanism to allow the operating system to selectively offload a subset of the established connections to the NIC [18, 22, 23]. Once a connection is handed off to the NIC, the operating system switches the protocol for that connection from TCP to a stateless bypass protocol that simply forwards application requests to send or receive data to the NIC. The NIC then performs all of the TCP processing for that connection. If necessary, the operating system can also reclaim the connection from the NIC after it has been handed off. Thus, the operating system retains complete control over the networking subsystem and can control the division of work between the NIC and host processor(s). At any time, the operating system can easily opt to reduce the number of connections on the NIC or not to use offload at all. Furthermore, with handoff, the NIC does not need to make routing decisions or allocate port numbers because established connections already have correct routes and ports.

While previous proposals have presented interfaces and implementations of TCP offload NICs that support connection handoff, they have not presented policies to utilize these NICs effectively. This paper shows that there are three main issues in systems that utilize connection handoff and evaluates policies to address these issues. First, the NIC must ensure that TCP processing on the NIC does not degrade the performance of connections that are being handled by the host. Second, the operating system must not overload the NIC since that would create a bottleneck in the system. Finally, when

the NIC is only able to store a limited number of connections, the operating system needs to hand off long-lived connections with high packet rates in order to better utilize the NIC.

Full-system simulations of four web workloads show that TCP processing on the NIC can degrade the performance of connections being handled by the host by slowing down packet deliveries to the host processor. The amount of time for a packet to cross the NIC increases from under 10 microseconds without handoff to over 1 millisecond with handoff. For three of the four workloads, the resulting request rate of the server is 14-30% lower than the baseline request rate achieved without handoff. The NIC can minimize delays by giving priority to those packets that must be delivered to the host. The use of this host first packet processing on the NIC increases the server's request rate over the baseline by up to 24%, when the NIC is not overloaded. However, the NIC still becomes overloaded when too many connections are handed off to the NIC, which reduces the request rate of the server by up to 44% below the baseline performance. The NIC can avoid overload conditions by dynamically adapting the number of connections to the current load indicated by the length of the receive packet queue. By using both techniques, handoff improves the request rate of the server by up to 31% over the baseline throughput. When the NIC can support a large number of connections, handing off connections in a simple first-come, first-served order is sufficient to realize these performance improvements. However, when the NIC has limited memory for storing connection state, handing off long-lived connections helps improve performance over a simple first-come, first-served handoff policy. Finally, using a NIC with a faster offload processor, handoff improves server throughput by 33-72%.

The rest of the paper is organized as follows. Section 2 briefly describes connection handoff. Section 3 presents techniques to control the division of work between the NIC and the operating system. Section 4 describes the experimental setup, and Section 5 presents results. Section 6 discusses related work. Section 7 draws conclusions.

2 Connection Handoff

TCP packet processing time is dominated by expensive main memory accesses, not computation [10, 17]. These main memory accesses occur when the network stack accesses packet data and connection data structures, which are rarely found in the processor caches. Accesses to packet data are due to data touching operations like data copies and checksum calculations. These accesses can be eliminated by using zero-copy I/O techniques to avoid data copies between the user and kernel

memory spaces [9, 12, 26] and checksum offload techniques to avoid computing TCP checksums on the host processor [19]. These techniques have gained wide acceptance in modern operating systems and network interfaces. However, no such techniques exist to eliminate accesses to connection data structures. While these structures are small, around 1KB per connection, a large number of connections can easily overwhelm modern processor caches, significantly degrading performance. Previous experiments show that main memory accesses to connection data structures can degrade performance as much as data touching operations [17].

TCP offload can be used to reduce the impact of expensive main memory accesses to connection data structures. By moving TCP processing to the NIC, connection data structures can be efficiently stored in fast, dedicated memories on the NIC. Typically, all TCP processing is moved to the NIC. However, such full TCP offload is not scalable. First, the resource limitations of a peripheral device will limit the maximum processing capability and memory capacity of a TCP offload NIC. Second, TCP offload complicates the existing software architecture of the network stack, since the operating system and the NIC now need to cooperatively manage global resources like port numbers and IP routes [24]. Connection handoff solves these problems by enabling the host operating system to select a subset of the established connections and move them to the network interface [18, 22, 23]. Using handoff, the operating system remains in control of global resources and can utilize TCP offload NICs to accelerate as many TCP connections as the resources on the NIC will allow.

Using connection handoff, all connections are established within the network stack of the host operating system. Then, if the operating system chooses to do so, the connection can be handed off to the NIC. Once a connection is handed off to the NIC, the NIC handles all TCP packets for that connection. Figure 1 shows a diagram of a network stack that supports connection handoff. The left portion of the diagram depicts the regular (non-offload) stack, and the dotted lines show data movement. When a connection is offloaded to the NIC, the operating system switches its stack to the right (shaded) portion of the diagram. This new stack executes a simple bypass protocol on the host processor for offloaded connections, and the rest of the TCP/IP stack is executed directly on the NIC. The bypass protocol forwards socket operations between the socket layers on the host and the NIC via the device driver, as shown by the dashed lines in the figure. For an offloaded connection, packets are generated by the NIC and also completely consumed by the NIC, so packet headers are never transferred across the local I/O interconnect. The solid lines show the packet movement within the NIC. The lookup layer on the NIC

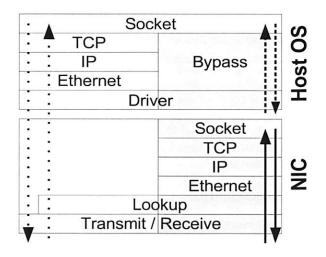


Figure 1: Modified network stack architecture to support connection handoff. The shaded region indicates the path used by connections on the NIC.

determines whether an incoming packet belongs to a connection on the NIC or to a connection on the host. It only adds a small amount of overhead to all incoming packets by using a hash-based lookup table [21].

The bypass layer communicates with the device driver using a connection handoff API. So, the operating system can transparently support multiple, heterogeneous NICs using the same API. Furthermore, the API also allows the actual socket buffer data to be stored in main memory in order to reduce the amount of buffering required on the NIC.

Previous experiments show that TCP offload, whether it is based on connection handoff or not, can reduce cycles, instructions, and cache misses on the host CPU as well as traffic across the local I/O interconnect [14, 18]. Since socket-level operations occur less frequently than Ethernet packet transmits and receives, handoff can reduce the number of message exchanges across the local I/O interconnect. For instance, because acknowledgment packets (ACKs) are processed by the NIC, the NIC may aggregate multiple ACKs into one message so that the host operating system can drop the acknowledged data in a single socket operation.

3 Connection Handoff Framework

In addition to the features described in the previous section, both the operating system and the network interface must also implement policies to ensure that the performance of connections that are being handled by the host operating system is not degraded, that the network interface does not become overloaded, and that the appropriate connections are handed off to the network interface. Figure 2 illustrates the proposed framework for

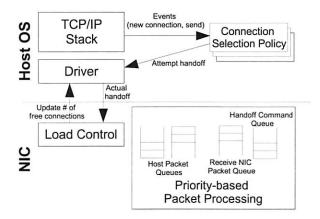


Figure 2: Framework for connection handoff and dynamically controlling the load on the NIC.

the implementation of such packet prioritization, load control, and connection selection policies. These policies are integral to the effective utilization of connection handoff network interfaces. The proposed framework and policies will be discussed in detail in the following sections—all supporting data was collected while running a simulated web server using the methodology that will be described in Section 4.

3.1 Priority-based Packet Processing

Traditional NICs act as a bridge between the network and main memory. There is very little processing required to send or receive packets from main memory (host packets), so the network interface can process each packet fairly quickly. However, with connection handoff, the NIC must perform TCP processing for packets that belong to connections that have been handed off to the network interface (NIC packets). Because NIC packets require significantly more processing than host packets, NIC packet processing can delay the processing of host packets, which may reduce the throughput of connections that remain within the host operating system. In the worst case, the NIC can become overloaded with TCP processing and drop host packets, which further reduces the throughput of host connections.

Table 1 illustrates the effect of connection handoff on host packet delays and overall networking performance for a web workload that uses 2048 simultaneous connections. The first row of the table shows the performance of the system when no connections are handed off to the NIC. In this case, the web server is able to satisfy 23031 requests per second with a median packet processing time on the NIC of only 2 us for host packets. The second row shows that when 256 of the 2048 connections (12.5%) are handed off to the NIC, the request rate increases by 4% with only slight increases in host packet processing time. However, as shown in the

		Host		NIC		
Offloaded	Packet	Packet Delay (usec) Idle (%		Idle (%)	Idle (%)	
Connections	Priority	Send	Receive	V-AV VII	, , , , , , , , , , , , , , , , , , ,	Requests/s
0	no handoff	2	2	0	62	23031
256	FCFS	3	3	0	49	23935
1024	FCFS	679	301	62	3	16121
1024	Host first	10	6	0	5	26663

Table 1: Impact of a heavily loaded NIC on the networking performance of a simulated web server. Packet delays are median values, not averages. Idle represents the fraction of total cycles that are idle.

third row, when 1024 connections are handed off to the NIC, the NIC is nearly saturated and becomes the bottleneck in the system. The median packet delay on the NIC for host packets increases dramatically and the NIC drops received packets as the receive queue fills up. As a result, the server's request rate drops by 33%.

For both the second and third rows of Table 1, the NIC processes all packets on a first-come, first-served (FCFS) basis. As the load on the NIC increases, host packets suffer increasing delays. Since an offloading NIC is doing additional work for NIC packets, increased host packet delays are inevitable. However, such delays need to be minimized in order to maintain the performance of host connections. Since host packets require very little processing on the NIC, they can be given a priority over NIC packets without significantly reducing the performance of connections that have been handed off to the NIC. The priority queues, shown in Figure 2, enable such a prioritization. As discussed in Section 2, all received packets must first go through the lookup layer, which determines whether a packet belongs to a connection on the NIC. Once the lookup task completes, the NIC now forms two queues. One queue includes only host packets, and the other queue stores only NIC packets. The NIC also maintains a queue of host packets to be sent and another queue of handoff command messages from the host. In order to give priority to host packets, the NIC always processes the queue of received host packets and the queue of host packets to be sent before NIC packets and handoff command messages.

The fourth row of Table 1 shows the impact of using the priority queues to implement a *host first* packet processing policy on the NIC. With the *host first* policy on the NIC, the median packet delay of host packets is about 6–10 us even though 1024 connections are handed off to the NIC. Handoff now results in a 16% improvement in request rate. Thus, this simple prioritization scheme can be an effective mechanism to ensure that TCP processing on the NIC does not hurt the performance of connections that are handled by the host operating system. Further evaluation is presented in Section 5.

3.2 Load Control

Packet prioritization can ensure that host packets are handled promptly, even when the NIC is heavily loaded. However, this will not prevent the NIC from becoming overloaded to the point where there are not enough processing resources remaining to process NIC packets. In such an overloaded condition, the network interface becomes a system bottleneck and degrades the performance of connections that have been handed off to the network interface. A weighted sum of the packet rate of NIC packets and the packet rate of host packets is an approximate measure of the load on the network interface. The number of connections that have been handed off to the NIC indirectly determines this load. In general, increasing the number of connections on the NIC increases the load on the NIC because they tend to increase overall packet rates. Likewise, decreasing the number of connections generally reduces the load on the NIC.

Due to the finite amount of memory on the NIC, there is a hard limit on the total number of connections that can be handed off to the NIC. However, depending on the workload and the available processing resources on the NIC, the NIC may become saturated well before the number of connections reaches the hard limit. Therefore, the network interface must dynamically control the number of connections that can be handed off based on the current load on the network interface.

As discussed in Section 3.1, the NIC maintains a queue of received NIC packets. As the load on the NIC increases, the NIC cannot service the receive queue as promptly. Therefore, the number of packets in the receive queue (queue length) is a good indicator of the current load on the NIC. This holds for send-dominated, receive-dominated, and balanced workloads. For senddominated workloads, the receive queue mainly stores acknowledgment packets. A large number of ACKs on the receive queue indicate that the load is too high because the host operating system is sending data much faster than the NIC can process ACKs returning from a remote machine. For receive-dominated workloads, the receive queue mainly stores data packets from remote machines. A large number of data packets on the receive queue indicates that data packets are being received much faster than the NIC can process and acknowledge them. In balanced workloads, a combination of the above factors will apply. Therefore, a large number of packets in the receive queue indicates that the NIC is not processing received packets in a timely manner which will increase packet delays for all connections.

The NIC uses six parameters to control the load on the network interface: Hard_limit, Soft_limit, Olen, Hiwat, Lowat, and Cnum. Hard_limit is the maximum possible number of connections that can be handed off to the network interface and is determined based on the amount of physical memory available on the network interface. Hard_limit is set when the network interface firmware is initialized and remains fixed. Soft_limit is the current maximum number of connections that may be handed off to the NIC. This parameter is initially set to Hard_limit, and is always less than or equal to Hard_limit. Qlen is the number of NIC packets currently on the receive packet queue. Hiwat is the high watermark for the receive packet queue. When Qlen exceeds Hiwat, the network interface is overloaded and must begin to reduce its load. A high watermark is needed because once the receive packet queue becomes full, packets will start to be dropped, so the load must be reduced before that point. Similarly, Lowat is the low watermark for the receive packet queue, indicating that the network interface is underloaded and should allow more connections to be handed off, if they are available. As with Hard_limit, Hiwat and Lowat are constants that are set upon initialization based upon the processing capabilities and memory capacity of the network interface. For example, a faster NIC with larger memory can absorb bursty traffic better than a slower NIC with smaller memory, so it should increase these values. Currently, Hiwat and Lowat need to be set empirically. However, since the values only depend on the hardware capabilities of the network interface, only the network interface manufacturer would need to tune the values, not the operating system. Finally, Cnum is the number of currently active connections on the NIC.

Figure 3 shows the state machine employed by the NIC in order to dynamically adjust the number of connections. The objective of the state machine is to maintain *Qlen* between *Lowat* and *Hiwat*. When *Qlen* grows above *Hiwat*, the NIC is assumed to be overloaded and should attempt to reduce the load by reducing *Soft_limit*. When *Qlen* drops below *Lowat*, the NIC is assumed to be underloaded and should attempt to increase the load by increasing *Soft_limit*.

The state machine starts in the MONITOR state. When *Qlen* becomes greater than *Hiwat*, the NIC reduces *Soft_limit*, sends a message to the device driver to advertise the new value, and transitions to the DECREASE state. While in the DECREASE state, the NIC waits

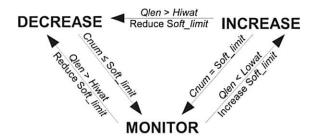


Figure 3: State machine used by the NIC firmware to dynamically control the number of connections on the NIC.

for connections to terminate. Once Cnum drops below Soft_limit, the state machine transitions back to the MON-ITOR state to assess the current load. If Qlen decreases below Lowat, then the NIC increases Soft_limit, sends a message to the device driver to notify it of the new Soft_limit, and transitions to the INCREASE state. In the INCREASE state, the NIC waits for new connections to arrive. If Cnum increases to Soft_limit, then the NIC transitions to the MONITOR state. However, while in the IN-CREASE state, if *Olen* increases above *Hiwat*, then the NIC reduces Soft_limit, sends a message to the device driver to alert it of the new value, and transitions to the DECREASE state. The state machine is simple and runs only when a packet arrives, the host hands off a new connection to the NIC, or an existing connection terminates. Thus, the run-time overhead of the state machine is insignificant.

As mentioned above, the NIC passively waits for connections to terminate while in the DECREASE state. Instead, the NIC may also actively restore the connections back to the host operating system and recover from an overload condition more quickly. The described framework easily supports such active restoration to the operating system. However, for this to be effective, the NIC would also need a mechanism to determine which connections are generating the most load, so should be restored first. Actively restoring connections in this manner was not necessary for the workloads studied in this paper, but it may help improve performance for other types of workloads.

While the receive packet queue length is easy to exploit, there are other measures of load such as idle time and packet rate. The network interface could calculate either of these metrics directly and use them to control the load. However, packet processing time is extremely dependent of the workload. Therefore, metrics such as the packet rate are difficult to use, as they are not directly related to the load on the NIC. This makes it more desirable to control the load based on resource use, such as the length of the receive queue or the idle time, than based on packet rate. The receive queue length was cho-

sen over idle time because it requires no NIC resources to compute.

3.3 Connection Selection

Whenever the network interface can handle additional connections, the operating system attempts to hand off established connections. The connection selection policy component of the framework depicted in Figure 2 decides whether the operating system should attempt to hand off a given connection. As described previously, the device driver then performs the actual handoff. The operating system may attempt handoff at any time after a connection is established. For instance, it may hand off a connection right after it is established, or when a packet is sent or received. If the handoff attempt fails, the operating system can try to handoff the connection again in the future. For simplicity, the current framework invokes the selection policy upon either connection establishments or send requests by the application and does not consider connections for handoff if the first handoff attempt for that connection fails.

The simplest connection selection policy is first-come, first-served. If all connections in the system have similar packet rates and lifetimes, then this is a reasonable choice, as all connections will benefit equally from offload. However, if connections in the system exhibit widely varying packet rates and lifetimes, then it is advantageous to consider the expected benefit of offloading a particular connection. These properties are highly dependent on the application, so a single selection policy may not perform well for all applications. Since applications typically use specific ports, the operating system should be able to employ multiple application-specific (per-port) connection selection policies.

Furthermore, the characteristics of the NIC can influence the types of connections that should be offloaded. Some offload processors may only be able to handle a small number of connections, but very quickly. For such offload processors, it is advantageous to hand off connections with high packet rates in order to fully utilize the processor. Other offload processors may have larger memory capacities, allowing them to handle a larger number of connections, but not as quickly. For these processors, it is more important to hand off as many connections as possible.

The expected benefit of handing off a connection is the packet processing savings over the lifetime of the connection minus the cost of the handoff. Here, the lifetime of a connection refers to the total number of packets sent and/or received through the connection. Therefore, it is clear that offloading a long-lived connection is more beneficial than a short-lived connection. The long-lived connection would accumulate enough per-packet savings to

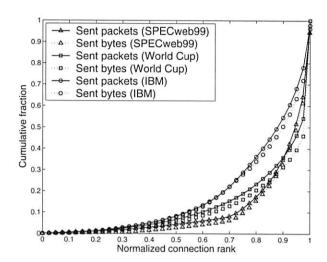


Figure 4: Distribution of connection lifetimes from SPECweb99 and the IBM and World Cup traces. Connection rank is based on the number of sent packets.

compensate for the handoff cost and also produce greater total saving than the short-lived connection during its lifetime.

In order for the operating system to compute the expected benefit of handing off a connection, it must be able to predict the connection's lifetime. Fortunately, certain workloads, such as web requests, show characteristic connection lifetime distributions, which can be used to predict a connection's lifetime. Figure 4 shows the distribution of connection lifetimes from several web workloads. The figure plots the cumulative fraction of sent packets and sent bytes of all connections over the length of the run. As shown in the figure, there are many short-lived connections, but the number of packets due to these connections account for a small fraction of total packets and bytes. For example, half of the connections are responsible for sending less than 10% of all packets for all three workloads. The other half of the connections send the remaining 90% of the packets. In fact, more than 45% of the total traffic is handled by less than 10% of the connections. The data shown in Figure 4 assumes that persistent connections are used. A persistent connection allows the client to reuse the connection for multiple requests. Persistent connections increase the average lifetime, but not the shape of distribution of lifetimes. Previous studies have shown that web workloads exhibit this kind of distribution [2, 7, 8]. The operating system may exploit such distribution in order to identify and hand off long-lived connections. For instance, since the number of packets transfered over a long-lived connection far exceeds that of a short connection, the system can use a threshold to differentiate long and short-lived connections. The operating system can simply keep track of the number of packets sent over a connection and hand

it off to the NIC only when the number reaches a certain threshold.

4 Experimental Setup

The authors have previously implemented connection handoff within FreeBSD 4.7 with the architecture described in Section 2 [18]. To evaluate the policies described in Section 3, this prototype is augmented with these policies using the existing framework. Since there are no offload controllers with open specifications, at least to the authors' best knowledge, an extended version of the full-system simulator Simics [20] is used for performance evaluations. Simics models the system hardware with enough detail that it can run complete and unmodified operating systems.

4.1 Simulation Setup

Simics is a functional full system simulator that allows the use of external modules to enforce timing. For the experiments, Simics has been extended with a memory system timing module and a network interface card module. The processor core is configured to execute one x86 instruction per cycle unless there are memory stalls. The memory system timing module includes a cycle accurate cache, memory controller, and DRAM simulator. All resource contention, latencies, and bandwidths within the memory controller and DRAM are accurately modeled [29]. Table 2 summarizes the simulator configuration.

The network interface simulator models a MIPS processor, 32 MB of memory, and several hardware components: PCI and Ethernet interfaces and a timer. The PCI and Ethernet interfaces provide direct memory access (DMA) and medium access control (MAC) capabilities, respectively. These are similar to those found on the Tigon programmable Gigabit Ethernet controller from Alteon [1]. Additionally, checksums are computed in hardware on the network interface. The firmware of the NIC uses these checksum values to support checksum offload for host packets and to avoid computing the checksums of NIC packets in software. The NIC does not employ any other hardware acceleration features such as hardware connection lookup tables [15]. The processor on the NIC runs the firmware and executes one instruction per cycle at a rate of 400, 600, or 800 million instructions per second (MIPS). The instruction rate is varied to evaluate the impact of NIC performance. Modern embedded processors are capable of such instruction rates with low power consumption [11]. At 400MIPS, the NIC can achieve 1Gb/s of TCP throughput for one offloaded connection and another 1Gb/s for a host connection simultaneously, using maximum-sized

	Configuration
CPU	Functional, single-issue, 2GHz x86 processor
	Instantaneous instruction fetch
L1 cache	64KB data cache
	Line size: 64B, associativity: 2-way
	Hit latency: 1 cycle
L2 cache	IMB data cache
	Line size: 64B, associativity: 16-way
	Hit latency: 15 cycles
	Prefetch: next-line on a miss
DRAM	DDR333 SDRAM of size 2GB
	Access latency: 195 cycles
NIC	Functional, single-issue processor
	Varied instruction rates for experiments
	Varied maximum number of connections
	10Gb/s wire

Table 2: Simulator configuration.

1518B Ethernet frames. The maximum number of connections that can be stored on the NIC is also varied in order to evaluate the impact of the amount of memory dedicated for storing connections. The network wire is set to run at 10Gb/s in order to eliminate the possibility of the wire being the bottleneck. The local I/O interconnect is not modeled due to its complexity. However, DMA transfers still correctly invalidate processor cache lines, as others have shown the importance of invalidations due to DMA [5].

The testbed consists of a server and a client machine, directly connected through a full-duplex 10Gb/s wire. Both are simulated using Simics. The server uses the configuration shown in Table 2, while the client is completely functional, so will never be a performance bottleneck.

4.2 Web Workloads

The experiments use SPECweb99 and two real web traces to drive the Flash web server [25]. SPECweb99 emulates multiple simultaneous clients. Each client issues requests for both static content (70%) and dynamic content (30%) and tries to maintain its bandwidth between 320Kb/s and 400Kb/s. The request sizes are statistically generated using a Zipf-like distribution in which a small number of files receive most of the requests. For static content, Flash sends HTTP response data through zero-copy I/O (the sendfile system call). All other types of data including HTTP headers and dynamically generated responses are copied between the user and kernel memory spaces.

The two web traces are from an IBM web site and the web site for the 1998 Soccer World Cup. A simple trace replayer program reads requests contained in the traces and sends them to the web server [4]. Like SPECweb99, the client program emulates multiple simultaneous clients. Unlike SPECweb99, it generates requests for static content only and sends new requests as fast as the server can handle. Both the replayer and SPECweb99 use persistent connections by default. The replayer uses a persistent connection for the requests from the same client that arrive within a fifteen second period in the given trace. SPECweb99 statistically chooses to use persistent connections for a fraction of all requests. To compare SPECweb99 against the two traces, the experiments also evaluate SPECweb99 that uses a non-default configuration where all requests are for static content.

For all experiments, the first 400000 packets are used to warm up the simulators, and measurements are taken during the next 600000 packets. Many recent studies based on simulations use purely functional simulators during the warmup phase to reduce simulation time. However, one recent study shows that such method can produce misleading results for TCP workloads and that the measurement phase needs to be long enough to cover several round trip times [16]. In this paper, the warmup phase simulates timing, and 600000 packets lead to at least one second of simulated time for the experiments presented in the paper.

5 Experimental Results

5.1 Priority-based Packet Processing and Load Control

Figure 5 shows the execution profiles of the simulated web server using various configurations. The Y axis shows abbreviated system configurations (see Table 3 for an explanation of the abbreviations). The first graph shows the fraction of host processor cycles spent in the user application, the operating system, and idle loop. The second graph shows the amount of idle time on the NIC. The third and fourth graphs show connection and packet rates. These graphs also show the fraction of connections that are handed off to the NIC, and the fraction of packets that are consumed and generated by the NIC while processing the connections on the NIC. The last two graphs show server throughput in requests/s, and HTTP content in megabits/s. HTTP content throughput only includes HTTP response bytes that are received by the client. Requests/s shows the request completion rates seen by the client.

W-0-N-N-FCFS-400 in Figure 5 shows the baseline performance of the simulated web server for the World Cup trace. No connections are handed off to the NIC. The host processor has zero idle time, and 57% of host processor cycles (not shown in the figure) are spent executing the network stack below the system call layer.

Configuration shorthands have the form
Workload-NIC Connections-Packet Priority-Load
Control-Selection Policy-NIC MIPS.

Control-Selection Policy-NIC MIPS.		
	Web server workload	
	W: World Cup, 2048 clients	
Workload	I: IBM, 2048 clients	
	D: SPECweb99, 2048 clients	
	S: SPECweb99-static, 4096 clients	
NIC	Maximum number of connections	
Connections	on the NIC	
Connections	0 means handoff is not used.	
	host first priority-based packet	
Packet	processing on the NIC	
Priority	P: Used	
	N: Not used	
Load	Load control mechanism on the NIC	
Control	L: Used	
Control	N: Not used	
	Connection selection policy used by	
Selection	the operating system	
Policy	FCFS: First-come, first-served	
	Tn: Threshold with value n	
NIC MIPS	Instruction rate of the NIC in million	
NIC MIFS	instructions per second	

Table 3: Configuration shorthands used in Section 5

Since the NIC has 62% idle time, handoff should be able to improve server performance. However, simply handing off many connection to the NIC can create a bottleneck at the NIC, as illustrated by *W-1024-N-N-FCFS-400*.

In W-1024-N-N-FCFS-400, the NIC can handle a maximum of 1024 connections at a time. 2048 connections are established, and 1024 of them are handed off to the NIC. As the NIC becomes nearly saturated with TCP processing (only 3% idle time), it takes too long to deliver host packets to the operating system. On average, it now takes more than 1 millisecond for a host packet to cross the NIC. Without handoff, it takes less than 10 microseconds. The 62% idle time on the host processor also shows that host packets are delivered too slowly. So, the connections on the NIC progress and terminate much faster than the connections on the host. When the client establishes new connections, they are most likely to replace terminated connections on the NIC, not the host. Consequently, the NIC processes a far greater share of new connections than the host. Overall, 88% of all connections during the experiment are handed off to the NIC. Note that at any given time, roughly half the active connections are being handled by the NIC and the other half are being handled by the host. Since the NIC becomes a bottleneck in the system and severely degrades the performance of connections handled by the

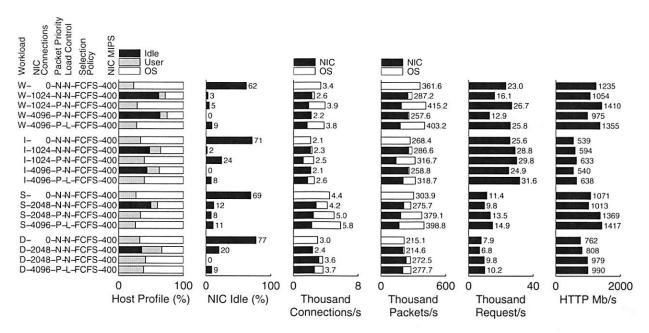


Figure 5: Impact of host first packet processing and load control on the simulated web server.

host, the request rate drops by 30%. This configuration clearly shows that naive offloading can degrade system performance. In *W-1024-P-N-FCFS-400*, the NIC still has a maximum of 1024 connections but employs *host first* packet processing to minimize delays to host packets. The mean time for a host packet to cross the NIC drops to less than 13 microseconds even though the NIC is still busy with TCP processing (only 5% idle time). The fraction of connections handed off to the NIC is now 48%, close to one half, as expected. The host processor shows no idle time, and server throughput continues to improve.

In W-4096-P-N-FCFS-400, the NIC can handle a maximum of 4096 connections at a time. 100% of connections are handed off to the NIC since there are only 2048 concurrent connections in the system. The NIC is fully saturated and again becomes a bottleneck in the system. Processing each packet takes much longer, and there are also dropped packets. As a result, the host processor shows 64% idle time, and the request rate drops by 52% from 26663/s to 12917/s. Thus, giving priority to host packets cannot prevent the NIC from becoming the bottleneck in the system. Note that host first packet processing still does it job, and host packets (mainly packets involved in new connection establishment) take only several microseconds to cross the NIC.

In W-4096-P-L-FCFS-400, the NIC can handle a maximum of 4096 connections at a time, just like W-4096-P-L-FCFS-400, but uses the load control mechanism discussed in Section 3.2. Figure 6 shows how the NIC dynamically adjusts the number of connections during the experiment. Initially 2048 connections are handled off to

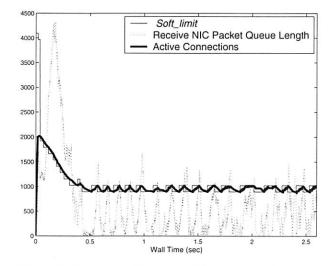


Figure 6: Dynamic adjustment of the number of connections on the NIC by the load control mechanism for configuration *W-4096-P-L-FCFS-400*.

the NIC, but received packets start piling up on the receive packet queue. As time progresses, the NIC reduces connections in order to keep the length of the receive packet queue under the threshold 1024. The number of connections on the NIC stabilizes around 1000 connections. The resulting server throughput is very close to that of *W-1024-P-N-FCFS-400* in which the NIC is manually set to handle up to 1024 concurrent connections. Thus, the load control mechanism is able to adjust the number of connections on the NIC in order to avoid overload conditions. The NIC now has 9% idle time, slightly greater than 5% shown in *W-1024-P-N-FCFS*-

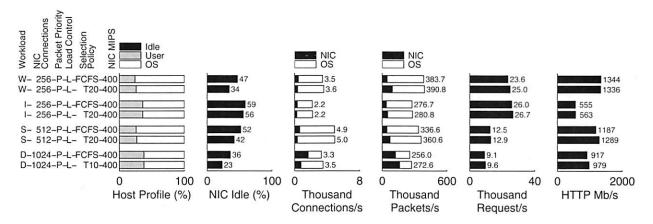


Figure 7: Impact of first-come, first-served and threshold-based connection selections on the simulated web server.

400, which indicates that the watermark values used in the load control mechanism are not optimal. Overall, handoff improves server throughput by 12% in packet rate, 12% in request rate, and 10% in HTTP content throughput (compare W-0-N-N-FCFS-400 and W-4096-P-L-FCFS-400). The server profiles during the execution of the IBM trace also show that both host first packet processing and the load control on the NIC are necessary, and that by using both techniques, handoff improves server throughput for the IBM trace by 19% in packet rate, 23% in request rate, and 18% in HTTP content throughput (compare I-0-N-N-FCFS-400 and I-4096-P-L-FCFS-400).

Unlike the trace replayer, SPECweb99 tries to maintain a fixed throughput for each client. Figure 5 also shows server performance for SPECweb99 Static and SPECweb99. The static version is same as SPECweb99 except that the client generates only static content requests, so it is used to compare against the results produced by the trace replayer. S-0-N-N-FCFS-400 shows the baseline performance for SPECweb99 Static. Since each client of SPECweb99 is throttled to a maximum of 400Kb/s, 4096 connections (twice the number used for the trace replayer) are used to saturate the server. Like W-0-N-N-FCFS-400, the host processor has no idle cycles and spends more than 70% of cycles in the kernel, and the NIC has 69% idle time. When 2048 connections are handed off, the request rate actually drops slightly. As in W-1024-N-N-FCFS-400, host packets are delivered to the operating system too slowly, and the host processor shows 50% idle time. The use of host first packet processing on the NIC overcomes this problem, and server throughput continues to increase. Increasing the number of connections further will simply overload the NIC as there is only 8% idle time. S-4096-P-L-FCFS-400 uses both host first packet processing and the load control mechanism on the NIC. Although the NIC can store all 4096 connections, the load control mechanism reduces the number of connections to around 2000 in order to avoid overload conditions. Overall, by using *host first* packet processing and the load control mechanism on the NIC, handoff improves the request rate for SPECweb99 Static by 31%. These techniques help improve server performance for regular SPECweb99 as well. Handoff improves the request rate by 28%.

5.2 Connection Selection Policy

As mentioned in Section 3.3, the system may use a threshold to differentiate long-lived connections that transfers many packets from short-lived ones. Handing off long-lived connections has the potential to improve server performance when the NIC has limited memory for a small number of connections. For instance, offload processors may use a small on-chip memory to store connections for fast access. In this case, it is necessary to be selective and hand off connections that transfer many packets in order to utilize the available compute power on the NIC as much as possible. On the other hand, when the NIC can handle a much larger number of connections, it is more important to hand off as many connections as possible, and a threshold-based selection policy has either negligible impact on server throughput or degrades it because fewer packets are processed by the NIC.

Figure 7 compares FCFS and threshold-based connection selection policies when the maximum number of connections on the NIC is much smaller than the value used in the previous section. For threshold-based policies, denoted by Tn, the trailing number indicates the minimum number of enqueue operations to the send socket buffer of a connection that must occur before the operating system attempts to hand off the connection. The number of enqueue operations is proportional to the number of sent packets. For instance, using T4, the operating system attempts a handoff when the fourth en-

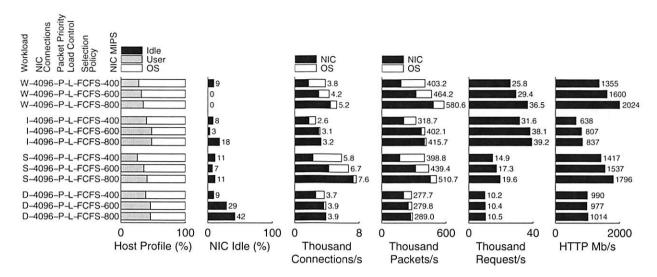


Figure 8: Impact of the instruction rate of the NIC on the simulated web server.

queue operation to the connection's send socket buffer occurs. As shown in the figure, the use of threshold enables the operating system to hand off longer connections than FCFS, but the resulting throughput improvements are small. For instance, W-256-P-L-FCFS-400 shows a case in which the NIC can handle up to 256 connections and the operating system hands off connections on a FCFS basis. 13% of connections and 12% of packets are processed by the NIC, as expected. The NIC shows 47% idle time. When a threshold policy is used (W-256-P-L-T20-400), the NIC now processes 24% of packets, and the request rate improves by 6%. However, the NIC still has 34% idle time. The lifetime distribution shown in Figure 4 suggests that if the operating system were able to pick longest 10% of connections, the NIC would process over 60% of packets. Thus, with a more accurate selection policy, the NIC would be able to process a greater fraction of packets and improve system performance further.

5.3 NIC Speed

The results so far have shown that the NIC must employ *host first* packet processing and dynamically control the number of connections. As the instruction rate of the NIC increases, the NIC processes packets more quickly. The load control mechanism on the NIC should be able to increase the number of connections handed off to the NIC. Figure 8 shows the impact of increasing the instruction rate of the NIC. *W-4096-P-L-FCFS-400* in the figure is same as the one in Figure 5 and is used as the baseline case. As the instruction rate increases from 400 to 600 and 800MIPS, the fraction of connections handed off to the NIC increases from 45% to 70% and 85%. Accordingly, the request rate of the server in-

creases from 25830/s to 29398/s and 36532/s (14% and 41% increases). For the IBM trace, increasing the instruction rate from 400 to 600MIPS results in a 21% increase in request rate. At 600MIPS, nearly all connections (95%) are handed off to the NIC. So, the faster 800MIPS NIC improves the request rate by only 3%.

NICs improve server throughput for SPECweb99 Static as well. As the instruction rate increases from 400 to 600MIPS, the request rate improves by 16%. The 800MIPS NIC further improves the request rate by 13%. Faster NICs do not benefit SPECweb99 because the 400MIPS NIC already achieves more than the specified throughput. With 2048 connections, SPECweb99 aims to achieve a maximum HTTP throughput of about $819\text{Mb/s} = 2048 \times 400\text{Kb/s}$. In reality, throughput can become greater than the specified rate as it is difficult to maintain throughput strictly under the specified rate. With the 400MIPS NIC, HTTP content throughput is near 1Gb/s. So, faster NICs simply have greater idle time.

These results show that the system can transparently exploit increased processing power on the NIC by using the load control mechanism and *host first* packet processing on the NIC. Thus, hardware developers can improve NIC capabilities without worrying about software changes as the firmware will adapt the number of connections and be able to use the increased processing power.

Finally, HTTP response times, measured as the amount of time elapsed between when a request is sent and when the full response is received, follow server request rates, as expected. For instance, the mean response time for the World Cup trace is 61ms without offload (W-0-N-N-FCFS-400). It increases to 99ms when 1024 connections are offloaded without host first packet processing or load control (W-1024-N-N-FCFS-400). The

use of both *host first* packet processing and load control drops the mean response time to 60ms (*W-1024-P-L-FCFS-400*). Increasing the instruction rate of the NIC from 400 to 600 and 800MIPS further reduces the mean response time to 53ms and 40ms, respectively. Mean response times for other workloads follow trends similar to that of the World Cup trace, except that mean response times for SPECweb99 are larger than those for the World Cup and IBM traces because of throttling and dynamic content generation.

6 Related Work

There are a number of previous studies on full TCP offload to both network interfaces and dedicated processors in the system. TCP servers was an early TCP offload design [27]. TCP servers, based on the Split-OS concept [3], splits TCP and the rest of the operating system and lets a dedicated processor or a dedicated system execute TCP. Brecht *et al.* expand this concept by providing an asynchronous I/O interface to communicate with the dedicated TCP processing resources [6]. Intel has dubbed such approaches, which dedicate a general-purpose processor to TCP processing, TCP *on*loading [28]. Regardless of the name, these approaches are effectively full TCP offload, as TCP and the rest of the system's processing are partitioned into two components.

Freimuth et al. recently showed that full offload reduces traffic on the local I/O interconnect [14]. They used two machines for evaluations, one acting as the NIC and the other as the host CPU. A central insight is that with offload, the NIC and the operating system communicate at a higher level than the conventional network interface, which gives opportunities for optimiza-Westrelin et al. also evaluated the impact of TCP offload [31]. They used a multiprocessor system in which one processor is dedicated to executing TCP, like TCP onloading, and show a significant improvement in microbenchmark performance. Finally, an analytical study on performance benefits of TCP offload shows that offload can be beneficial but its benefits can vary widely depending on application and hardware characteristics [30].

However, while these studies have shown the benefits of TCP offload, they have not addressed the problems that have been associated with full TCP offload. These problems include creating a potential bottleneck at the NIC, difficulties in designing software interfaces between the operating system and the NIC, modifying the existing network stack implementations, and introducing a new source of software bugs (at the NIC) [24].

Connection handoff, which addresses some of these concerns, has been previously proposed and implemented. Microsoft has proposed to implement a device driver API for TCP offload NICs based on connection handoff in the next generation Windows operating system, as part of the Chimney Offload architecture [22]. Mogul et al. argued that exposing transport (connection) states to the application creates opportunities for enhanced application features and performance optimizations, including moving connection states between the operating system and offload NICs [23]. The authors have implemented both the operating system and network interface components of connection handoff, with the architecture described in Section 2 [18]. The policies presented in this paper apply to all of these previous proposals and implementations and will improve their efficiency and performance, and prevent the network interface from becoming a performance bottleneck.

A recent study shows that a commercial offloading NIC can achieve over 7Gb/s and substantially improve web server throughput [13]. This is an encouraging result since it shows that a specialized offload processor can handle high packet rates.

7 Conclusion

Offloading TCP processing to the NIC can improve system throughput by reducing computation and memory bandwidth requirements on the host processor. However, the NIC inevitably has limited resources and can become a bottleneck in the system. Offload based on connection handoff enables the operating system to control the number of connections processed by the host processor and the NIC, thereby controlling the division of work between them. Thus, the system should be able to treat the NIC as an acceleration coprocessor by handing off as many connections as the resources on the NIC will allow.

A system that implements connection handoff can employ the policies presented in this paper in order to fully utilize the offload NIC without creating a bottleneck in the system. First, the NIC gives priority to those packets that belong to the connections processed by the host processor. This ensures that packets are delivered to the operating system in timely manner and that TCP processing on the NIC does not degrade the performance of host connections. Second, the NIC dynamically controls the number of connections that can be handed off. This avoids overloading the NIC, which would create a performance bottleneck in the system. Third, the operating system can differentiate connections and hand off only long-lived connections to the NIC in order to better utilize offloading NICs that lack memory capacity for a large number of connections. Full-system simulations of web workloads show that without any of the policies handoff reduces the server request rate by up to 44%. In contrast, connection handoff augmented with these polices successfully improves server request rate by 12–31%. When a faster offload processor is used, the system transparently exploits the increased processing capacity of the NIC, and connection handoff achieves request rates that are 33-72% higher than a system without handoff.

Acknowledgments

The authors thank Alan L. Cox for his interest and comments on the paper. The authors also thank Robbert van Renesse for shepherding and the reviewers for their valuable comments. This work is supported in part by a donation from Advanced Micro Devices and by the National Science Foundation under Grant Nos. CCR-0209174 and CCF-0546140.

References

- Alteon Networks. Tigon/PCI Ethernet Controller, Aug. 1997. Revision 1.04.
- [2] M. F. Arlitt and C. L. Williamson. Internet Web Servers: Workload Characterization and Performance Implications. *IEEE/ACM Transactions on Networking*, 5(5):631– 645, Oct. 1997.
- [3] K. Banerjee, A. Bohra, S. Gopalakrishnan, M. Rangarajan, and L. Iftode. Split-OS: An Operating System Architecture for Clusters of Intelligent Devices. Work-in-Progress Session at the 18th Symposium on Operating Systems Principles, Oct. 2001.
- [4] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In Proceedings of the USENIX Symposium on Internet Technologies and Systems, Dec. 1997.
- [5] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-Oriented Full-System Simulation using M5. In Proceedings of the Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW), pages 36–43, Feb. 2003.
- [6] T. Brecht, G. J. Janakiraman, B. Lynn, V. Saletore, and Y. Turner. Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O. In *Proceedings of EuroSys* 2006, pages 265–278, Apr. 2006.
- [7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Schenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE INFOCOM* '99, volume 1, pages 126–134, Mar. 1999.
- [8] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the 1997 USENIX Sym*posium on Internet Technology and Systems, pages 193– 206, Dec. 1997.
- [9] H. K. J. Chu. Zero-Copy TCP in Solaris. In *Proceedings* of the 1996 Annual USENIX Technical Conference, pages 253–264, 1996.
- [10] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communi*cations Magazine, pages 23–29, June 1989.
- [11] L. T. Clark, E. J. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M. Morrow, K. E. Velarde, and M. A. Yarch. An Embedded 32-b Microprocessor Core for Low-Power and High-Performance Applications. *IEEE Journal of Solid-State Circuits*, 36(11):1599–1608, Nov. 2001.

- [12] P. Druschel and L. L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the* 14th Symposium on Operating Systems Principles (SOSP-14), pages 189–202, Dec. 1993.
- [13] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda. Performance Characterization of a 10-Gigabit Ethernet TOE. In *Proceedings of the 13th IEEE Sym*posium on High-Performance Interconnects, 2005.
- [14] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey. Server Network Scalability and TCP Offload. In *Proceedings of the 2005 Annual USENIX Technical Conference*, pages 209–222, Apr. 2005.
- [15] Y. Hoskote, B. A. Bloechel, G. E. Dermer, V. Erraguntla, D. Finan, J. Howard, D. Klowden, S. Narendra, G. Ruhl, J. W. Tschanz, S. Vangal, V. Veeramachaneni, H. Wilson, J. Xu, and N. Borkar. A TCP Offload Accelerator for 10 Gb/s Ethernet in 90-nm CMOS. *IEEE Journal of Solid-State Circuits*, 38(11):1866–1875, Nov. 2003.
- [16] L. R. Hsu, A. G. Saidi, N. L. Binkert, and S. K. Reinhardt. Sampling and Stability in TCP/IP Workloads. In *Proceedings of the First Annual Workshop on Modeling, Benchmarking, and Simulation (MoBS)*, pages 68–77, 2005.
- [17] H. Kim and S. Rixner. Performance Characterization of the FreeBSD Network Stack. Computer Science Department, Rice University, June 2005. Technical Report TR05-450.
- [18] H. Kim and S. Rixner. TCP Offload through Connection Handoff. In *Proceedings of EuroSys* 2006, pages 279– 290, Apr. 2006.
- [19] K. Kleinpaste, P. Steenkiste, and B. Zill. Software Support for Outboard Buffering and Checksumming. In Proceedings of the ACM SIGCOMM '95 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, pages 87–98, Aug. 1995.
- [20] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hállberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.
- [21] P. E. McKenney and K. F. Dove. Efficient Demultiplexing of Incoming TCP Packets. In Proceedings of the ACM SIGCOMM '92 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, pages 269–279, 1992.
- [22] Microsoft Corporation. Scalable Networking: Network Protocol Offload – Introducing TCP Chimney, Apr. 2004. WinHEC Version.
- [23] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore. Unveiling the Transport. ACM SIGCOMM Computer Communication Review, 34(1):99–106, 2004.
- [24] J. C. Mogul. TCP offload is a dumb idea whose time has come. In Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems, pages 25–30, 2003.
- [25] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the* USENIX 1999 Annual Technical Conference, pages 199– 212, June 1999.
- [26] V. S. Pai, P. Druschel, and W. Zwaenepoel. I/O-Lite: A Unified I/O Buffering and Caching System. In Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation, pages 15–28, Feb. 1999.
- [27] M. Rangarajan, A. Bohra, K. Banerjee, E. V. Carrera, R. Bianchini, L. Iftode, and W. Zwaenepoel. TCP Servers: Offloading TCP/IP Processing in Internet Servers. Design, Implementation, and Performance. Computer Science Department, Rutgers University, Mar. 2002. Technical Report DCR-TR-481.

- [28] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP Onloading for Data Center Servers. *Computer*, 37(11):48–58, Nov. 2004.
- [29] S. Rixner. Memory Controller Optimizations for Web Servers. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 355–366, Dec. 2004.
- [30] P. Shivam and J. S. Chase. On the Elusive Benefits of Protocol Offload. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence*, pages 179–184, 2003.
- [31] R. Westrelin, N. Fugier, E. Nordmark, K. Kunze, and E. Lemoine. Studying Network Protocol Offload With Emulation: Approach And Preliminary Results. In *Proceedings of the 12th Annual IEEE Symposium on High Performance Interconnects*, pages 84–90, Aug. 2004.

Ceph: A Scalable, High-Performance Distributed File System

Sage A. Weil

Scott A. Brandt Ethan L. Miller Carlos Maltzahn

Darrell D. E. Long

University of California, Santa Cruz {sage, scott, elm, darrell, carlosm}@cs.ucsc.edu

Abstract

We have developed Ceph, a distributed file system that provides excellent performance, reliability, and scalability. Ceph maximizes the separation between data and metadata management by replacing allocation tables with a pseudo-random data distribution function (CRUSH) designed for heterogeneous and dynamic clusters of unreliable object storage devices (OSDs). We leverage device intelligence by distributing data replication, failure detection and recovery to semi-autonomous OSDs running a specialized local object file system. A dynamic distributed metadata cluster provides extremely efficient metadata management and seamlessly adapts to a wide range of general purpose and scientific computing file system workloads. Performance measurements under a variety of workloads show that Ceph has excellent I/O performance and scalable metadata management, supporting more than 250,000 metadata operations per second.

1 Introduction

System designers have long sought to improve the performance of file systems, which have proved critical to the overall performance of an exceedingly broad class of applications. The scientific and high-performance computing communities in particular have driven advances in the performance and scalability of distributed storage systems, typically predicting more general purpose needs by a few years. Traditional solutions, exemplified by NFS [20], provide a straightforward model in which a server exports a file system hierarchy that clients can map into their local name space. Although widely used, the centralization inherent in the client/server model has proven a significant obstacle to scalable performance.

More recent distributed file systems have adopted architectures based on object-based storage, in which conventional hard disks are replaced with intelligent object storage devices (OSDs) which combine a CPU, network interface, and local cache with an underlying disk or RAID [4, 7, 8, 32, 35]. OSDs replace the traditional block-level interface with one in which clients can read or write byte ranges to much larger (and often variably sized) named objects, distributing low-level block allocation decisions to the devices themselves. Clients typically interact with a metadata server (MDS) to perform metadata operations (*open*, *rename*), while communicating directly with OSDs to perform file I/O (reads and writes), significantly improving overall scalability.

Systems adopting this model continue to suffer from scalability limitations due to little or no distribution of the metadata workload. Continued reliance on traditional file system principles like allocation lists and inode tables and a reluctance to delegate intelligence to the OSDs have further limited scalability and performance, and increased the cost of reliability.

We present Ceph, a distributed file system that provides excellent performance and reliability while promising unparalleled scalability. Our architecture is based on the assumption that systems at the petabyte scale are inherently dynamic: large systems are inevitably built incrementally, node failures are the norm rather than the exception, and the quality and character of workloads are constantly shifting over time.

Ceph decouples data and metadata operations by eliminating file allocation tables and replacing them with generating functions. This allows Ceph to leverage the intelligence present in OSDs to distribute the complexity surrounding data access, update serialization, replication and reliability, failure detection, and recovery. Ceph utilizes a highly adaptive distributed metadata cluster architecture that dramatically improves the scalability of metadata access, and with it, the scalability of the entire system. We discuss the goals and workload assumptions motivating our choices in the design of the architecture, analyze their impact on system scalability and performance, and relate our experiences in implementing a functional system prototype.

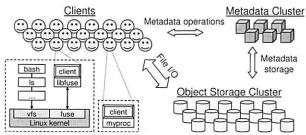


Figure 1: System architecture. Clients perform file I/O by communicating directly with OSDs. Each process can either link directly to a client instance or interact with a mounted file system.

2 System Overview

The Ceph file system has three main components: the client, each instance of which exposes a near-POSIX file system interface to a host or process; a cluster of OSDs, which collectively stores all data and metadata; and a metadata server cluster, which manages the namespace (file names and directories) while coordinating security, consistency and coherence (see Figure 1). We say the Ceph interface is near-POSIX because we find it appropriate to extend the interface and selectively relax consistency semantics in order to better align with the needs of applications and to improve system performance.

The primary goals of the architecture are scalability (to hundreds of petabytes and beyond), performance, and reliability. Scalability is considered in a variety of dimensions, including the overall storage capacity and throughput of the system, and performance in terms of individual clients, directories, or files. Our target workload may include such extreme cases as tens or hundreds of thousands of hosts concurrently reading from or writing to the same file or creating files in the same directory. Such scenarios, common in scientific applications running on supercomputing clusters, are increasingly indicative of tomorrow's general purpose workloads. More importantly, we recognize that distributed file system workloads are inherently dynamic, with significant variation in data and metadata access as active applications and data sets change over time. Ceph directly addresses the issue of scalability while simultaneously achieving high performance, reliability and availability through three fundamental design features: decoupled data and metadata, dynamic distributed metadata management, and reliable autonomic distributed object storage.

Decoupled Data and Metadata—Ceph maximizes the separation of file metadata management from the storage of file data. Metadata operations (*open*, *rename*, etc.) are collectively managed by a metadata server cluster, while clients interact directly with OSDs to perform file I/O (reads and writes). Object-based storage has long promised to improve the scalability of file systems by

delegating low-level block allocation decisions to individual devices. However, in contrast to existing object-based file systems [4, 7, 8, 32] which replace long per-file block lists with shorter object lists, Ceph eliminates allocation lists entirely. Instead, file data is striped onto predictably named objects, while a special-purpose data distribution function called CRUSH [29] assigns objects to storage devices. This allows any party to calculate (rather than look up) the name and location of objects comprising a file's contents, eliminating the need to maintain and distribute object lists, simplifying the design of the system, and reducing the metadata cluster workload.

Dynamic Distributed Metadata Management— Because file system metadata operations make up as much as half of typical file system workloads [22], effective metadata management is critical to overall system performance. Ceph utilizes a novel metadata cluster architecture based on Dynamic Subtree Partitioning [30] that adaptively and intelligently distributes responsibility for managing the file system directory hierarchy among tens or even hundreds of MDSs. A (dynamic) hierarchical partition preserves locality in each MDS's workload, facilitating efficient updates and aggressive prefetching to improve performance for common workloads. Significantly, the workload distribution among metadata servers is based entirely on current access patterns, allowing Ceph to effectively utilize available MDS resources under any workload and achieve near-linear scaling in the number of MDSs.

Reliable Autonomic Distributed Object Storage— Large systems composed of many thousands of devices are inherently dynamic: they are built incrementally, they grow and contract as new storage is deployed and old devices are decommissioned, device failures are frequent and expected, and large volumes of data are created, moved, and deleted. All of these factors require that the distribution of data evolve to effectively utilize available resources and maintain the desired level of data replication. Ceph delegates responsibility for data migration, replication, failure detection, and failure recovery to the cluster of OSDs that store the data, while at a high level, OSDs collectively provide a single logical object store to clients and metadata servers. This approach allows Ceph to more effectively leverage the intelligence (CPU and memory) present on each OSD to achieve reliable, highly available object storage with linear scaling.

We describe the operation of the Ceph client, metadata server cluster, and distributed object store, and how they are affected by the critical features of our architecture. We also describe the status of our prototype.

3 Client Operation

We introduce the overall operation of Ceph's components and their interaction with applications by describ-

ing Ceph's client operation. The Ceph client runs on each host executing application code and exposes a file system interface to applications. In the Ceph prototype, the client code runs entirely in user space and can be accessed either by linking to it directly or as a mounted file system via FUSE [25] (a user-space file system interface). Each client maintains its own file data cache, independent of the kernel page or buffer caches, making it accessible to applications that link to the client directly.

3.1 File I/O and Capabilities

When a process opens a file, the client sends a request to the MDS cluster. An MDS traverses the file system hierarchy to translate the file name into the file inode, which includes a unique inode number, the file owner, mode, size, and other per-file metadata. If the file exists and access is granted, the MDS returns the inode number, file size, and information about the striping strategy used to map file data into objects. The MDS may also issue the client a capability (if it does not already have one) specifying which operations are permitted. Capabilities currently include four bits controlling the client's ability to read, cache reads, write, and buffer writes. In the future, capabilities will include security keys allowing clients to prove to OSDs that they are authorized to read or write data [13, 19] (the prototype currently trusts all clients). Subsequent MDS involvement in file I/O is limited to managing capabilities to preserve file consistency and achieve proper semantics.

Ceph generalizes a range of striping strategies to map file data onto a sequence of objects. To avoid any need for file allocation metadata, object names simply combine the file inode number and the stripe number. Object replicas are then assigned to OSDs using CRUSH, a globally known mapping function (described in Section 5.1). For example, if one or more clients open a file for read access, an MDS grants them the capability to read and cache file content. Armed with the inode number, layout, and file size, the clients can name and locate all objects containing file data and read directly from the OSD cluster. Any objects or byte ranges that don't exist are defined to be file "holes," or zeros. Similarly, if a client opens a file for writing, it is granted the capability to write with buffering, and any data it generates at any offset in the file is simply written to the appropriate object on the appropriate OSD. The client relinquishes the capability on file close and provides the MDS with the new file size (the largest offset written), which redefines the set of objects that (may) exist and contain file data.

3.2 Client Synchronization

POSIX semantics sensibly require that reads reflect any data previously written, and that writes are atomic (*i. e.*, the result of overlapping, concurrent writes will reflect a particular order of occurrence). When a file is opened by

multiple clients with either multiple writers or a mix of readers and writers, the MDS will revoke any previously issued read caching and write buffering capabilities, forcing client I/O for that file to be synchronous. That is, each application read or write operation will block until it is acknowledged by the OSD, effectively placing the burden of update serialization and synchronization with the OSD storing each object. When writes span object boundaries, clients acquire exclusive locks on the affected objects (granted by their respective OSDs), and immediately submit the write and unlock operations to achieve the desired serialization. Object locks are similarly used to mask latency for large writes by acquiring locks and flushing data asynchronously.

Not surprisingly, synchronous I/O can be a performance killer for applications, particularly those doing small reads or writes, due to the latency penalty—at least one round-trip to the OSD. Although read-write sharing is relatively rare in general-purpose workloads [22], it is more common in scientific computing applications [27], where performance is often critical. For this reason, it is often desirable to relax consistency at the expense of strict standards conformance in situations where applications do not rely on it. Although Ceph supports such relaxation via a global switch, and many other distributed file systems punt on this issue [20], this is an imprecise and unsatisfying solution: either performance suffers, or consistency is lost system-wide.

For precisely this reason, a set of high performance computing extensions to the POSIX I/O interface have been proposed by the high-performance computing (HPC) community [31], a subset of which are implemented by Ceph. Most notably, these include an O_LAZY flag for open that allows applications to explicitly relax the usual coherency requirements for a shared-write file. Performance-conscious applications which manage their own consistency (e.g., by writing to different parts of the same file, a common pattern in HPC workloads [27]) are then allowed to buffer writes or cache reads when I/O would otherwise be performed synchronously. If desired, applications can then explicitly synchronize with two additional calls: lazyio_propagate will flush a given byte range to the object store, while lazyio_synchronize will ensure that the effects of previous propagations are reflected in any subsequent reads. The Ceph synchronization model thus retains its simplicity by providing correct read-write and shared-write semantics between clients via synchronous I/O, and extending the application interface to relax consistency for performance conscious distributed applications.

3.3 Namespace Operations

Client interaction with the file system namespace is managed by the metadata server cluster. Both read operations

(e. g., readdir, stat) and updates (e. g., unlink, chmod) are synchronously applied by the MDS to ensure serialization, consistency, correct security, and safety. For simplicity, no metadata locks or leases are issued to clients. For HPC workloads in particular, callbacks offer minimal upside at a high potential cost in complexity.

Instead, Ceph optimizes for the most common metadata access scenarios. A readdir followed by a stat of each file (e.g., 1s -1) is an extremely common access pattern and notorious performance killer in large directories. A readdir in Ceph requires only a single MDS request, which fetches the entire directory, including inode contents. By default, if a readdir is immediately followed by one or more stats, the briefly cached information is returned; otherwise it is discarded. Although this relaxes coherence slightly in that an intervening inode modification may go unnoticed, we gladly make this trade for vastly improved performance. This behavior is explicitly captured by the readdirplus [31] extension, which returns lstat results with directory entries (as some OS-specific implementations of getdir already do).

Ceph could allow consistency to be further relaxed by caching metadata longer, much like earlier versions of NFS, which typically cache for 30 seconds. However, this approach breaks coherency in a way that is often critical to applications, such as those using *stat* to determine if a file has been updated—they either behave incorrectly, or end up waiting for old cached values to time out.

We opt instead to again provide correct behavior and extend the interface in instances where it adversely affects performance. This choice is most clearly illustrated by a stat operation on a file currently opened by multiple clients for writing. In order to return a correct file size and modification time, the MDS revokes any write capabilities to momentarily stop updates and collect up-todate size and mtime values from all writers. The highest values are returned with the stat reply, and capabilities are reissued to allow further progress. Although stopping multiple writers may seem drastic, it is necessary to ensure proper serializability. (For a single writer, a correct value can be retrieved from the writing client without interrupting progress.) Applications for which coherent behavior is unnecessary—victims of a POSIX interface that doesn't align with their needs—can use statlite [31], which takes a bit mask specifying which inode fields are not required to be coherent.

4 Dynamically Distributed Metadata

Metadata operations often make up as much as half of file system workloads [22] and lie in the critical path, making the MDS cluster critical to overall performance. Metadata management also presents a critical scaling challenge in distributed file systems: although capacity and aggregate I/O rates can scale almost arbitrarily with the

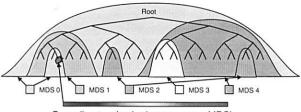
addition of more storage devices, metadata operations involve a greater degree of interdependence that makes scalable consistency and coherence management more difficult.

File and directory metadata in Ceph is very small, consisting almost entirely of directory entries (file names) and inodes (80 bytes). Unlike conventional file systems, no file allocation metadata is necessary—object names are constructed using the inode number, and distributed to OSDs using CRUSH. This simplifies the metadata workload and allows our MDS to efficiently manage a very large working set of files, independent of file sizes. Our design further seeks to minimize metadata related disk I/O through the use of a two-tiered storage strategy, and to maximize locality and cache efficiency with Dynamic Subtree Partitioning [30].

4.1 Metadata Storage

Although the MDS cluster aims to satisfy most requests from its in-memory cache, metadata updates must be committed to disk for safety. A set of large, bounded, lazily flushed journals allows each MDS to quickly stream its updated metadata to the OSD cluster in an efficient and distributed manner. The per-MDS journals, each many hundreds of megabytes, also absorb repetitive metadata updates (common to most workloads) such that when old journal entries are eventually flushed to long-term storage, many are already rendered obsolete. Although MDS recovery is not yet implemented by our prototype, the journals are designed such that in the event of an MDS failure, another node can quickly rescan the journal to recover the critical contents of the failed node's in-memory cache (for quick startup) and in doing so recover the file system state.

This strategy provides the best of both worlds: streaming updates to disk in an efficient (sequential) fashion, and a vastly reduced re-write workload, allowing the long-term on-disk storage layout to be optimized for future read access. In particular, inodes are embedded directly within directories, allowing the MDS to prefetch entire directories with a single OSD read request and exploit the high degree of directory locality present in most workloads [22]. Each directory's content is written to the OSD cluster using the same striping and distribution strategy as metadata journals and file data. Inode numbers are allocated in ranges to metadata servers and considered immutable in our prototype, although in the future they could be trivially reclaimed on file deletion. An auxiliary anchor table [28] keeps the rare inode with multiple hard links globally addressable by inode number—all without encumbering the overwhelmingly common case of singly-linked files with an enormous, sparsely populated and cumbersome inode table.



Busy directory hashed across many MDS's

Figure 2: Ceph dynamically maps subtrees of the directory hierarchy to metadata servers based on the current workload. Individual directories are hashed across multiple nodes only when they become hot spots.

4.2 Dynamic Subtree Partitioning

Our primary-copy caching strategy makes a single authoritative MDS responsible for managing cache coherence and serializing updates for any given piece of metadata. While most existing distributed file systems employ some form of static subtree-based partitioning to delegate this authority (usually forcing an administrator to carve the dataset into smaller static "volumes"), some recent and experimental file systems have used hash functions to distribute directory and file metadata [4], effectively sacrificing locality for load distribution. Both approaches have critical limitations: static subtree partitioning fails to cope with dynamic workloads and data sets, while hashing destroys metadata locality and critical opportunities for efficient metadata prefetching and storage.

Ceph's MDS cluster is based on a dynamic subtree partitioning strategy [30] that adaptively distributes cached metadata hierarchically across a set of nodes, as illustrated in Figure 2. Each MDS measures the popularity of metadata within the directory hierarchy using counters with an exponential time decay. Any operation increments the counter on the affected inode and all of its ancestors up to the root directory, providing each MDS with a weighted tree describing the recent load distribution. MDS load values are periodically compared, and appropriately-sized subtrees of the directory hierarchy are migrated to keep the workload evenly distributed. The combination of shared long-term storage and carefully constructed namespace locks allows such migrations to proceed by transferring the appropriate contents of the in-memory cache to the new authority, with minimal impact on coherence locks or client capabilities. Imported metadata is written to the new MDS's journal for safety, while additional journal entries on both ends ensure that the transfer of authority is invulnerable to intervening failures (similar to a two-phase commit). The resulting subtree-based partition is kept coarse to minimize prefix replication overhead and to preserve locality.

When metadata is replicated across multiple MDS nodes, inode contents are separated into three groups, each with different consistency semantics: security

(owner, mode), file (size, mtime), and immutable (inode number, ctime, layout). While immutable fields never change, security and file locks are governed by independent finite state machines, each with a different set of states and transitions designed to accommodate different access and update patterns while minimizing lock contention. For example, owner and mode are required for the security check during path traversal but rarely change, requiring very few states, while the file lock reflects a wider range of client access modes as it controls an MDS's ability to issue client capabilities.

4.3 Traffic Control

Partitioning the directory hierarchy across multiple nodes can balance a broad range of workloads, but cannot always cope with hot spots or flash crowds, where many clients access the same directory or file. Ceph uses its knowledge of metadata popularity to provide a wide distribution for hot spots only when needed and without incurring the associated overhead and loss of directory locality in the general case. The contents of heavily read directories (e.g., many opens) are selectively replicated across multiple nodes to distribute load. Directories that are particularly large or experiencing a heavy write workload (e.g., many file creations) have their contents hashed by file name across the cluster, achieving a balanced distribution at the expense of directory locality. This adaptive approach allows Ceph to encompass a broad spectrum of partition granularities, capturing the benefits of both coarse and fine partitions in the specific circumstances and portions of the file system where those strategies are most effective.

Every MDS response provides the client with updated information about the authority and any replication of the relevant inode and its ancestors, allowing clients to learn the metadata partition for the parts of the file system with which they interact. Future metadata operations are directed at the authority (for updates) or a random replica (for reads) based on the deepest known prefix of a given path. Normally clients learn the locations of unpopular (unreplicated) metadata and are able to contact the appropriate MDS directly. Clients accessing popular metadata, however, are told the metadata reside either on different or multiple MDS nodes, effectively bounding the number of clients believing any particular piece of metadata resides on any particular MDS, dispersing potential hot spots and flash crowds before they occur.

5 Distributed Object Storage

From a high level, Ceph clients and metadata servers view the object storage cluster (possibly tens or hundreds of thousands of OSDs) as a single logical object store and namespace. Ceph's Reliable Autonomic Distributed Object Store (RADOS) achieves linear scaling in both

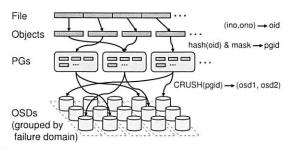


Figure 3: Files are striped across many objects, grouped into *placement groups* (PGs), and distributed to OSDs via CRUSH, a specialized replica placement function.

capacity and aggregate performance by delegating management of object replication, cluster expansion, failure detection and recovery to OSDs in a distributed fashion.

5.1 Data Distribution with CRUSH

Ceph must distribute petabytes of data among an evolving cluster of thousands of storage devices such that device storage and bandwidth resources are effectively utilized. In order to avoid imbalance (e. g., recently deployed devices mostly idle or empty) or load asymmetries (e. g., new, hot data on new devices only), we adopt a strategy that distributes new data randomly, migrates a random subsample of existing data to new devices, and uniformly redistributes data from removed devices. This stochastic approach is robust in that it performs equally well under any potential workload.

Ceph first maps objects into placement groups (PGs) using a simple hash function, with an adjustable bit mask to control the number of PGs. We choose a value that gives each OSD on the order of 100 PGs to balance variance in OSD utilizations with the amount of replicationrelated metadata maintained by each OSD. Placement groups are then assigned to OSDs using CRUSH (Controlled Replication Under Scalable Hashing) [29], a pseudo-random data distribution function that efficiently maps each PG to an ordered list of OSDs upon which to store object replicas. This differs from conventional approaches (including other object-based file systems) in that data placement does not rely on any block or object list metadata. To locate any object, CRUSH requires only the placement group and an OSD cluster map: a compact, hierarchical description of the devices comprising the storage cluster. This approach has two key advantages: first, it is completely distributed such that any party (client, OSD, or MDS) can independently calculate the location of any object; and second, the map is infrequently updated, virtually eliminating any exchange of distribution-related metadata. In doing so, CRUSH simultaneously solves both the data distribution problem ("where should I store data") and the data location problem ("where did I store data"). By design, small changes

to the storage cluster have little impact on existing PG mappings, minimizing data migration due to device failures or cluster expansion.

The cluster map hierarchy is structured to align with the clusters physical or logical composition and potential sources of failure. For instance, one might form a fourlevel hierarchy for an installation consisting of shelves full of OSDs, rack cabinets full of shelves, and rows of cabinets. Each OSD also has a weight value to control the relative amount of data it is assigned. CRUSH maps PGs onto OSDs based on placement rules, which define the level of replication and any constraints on placement. For example, one might replicate each PG on three OSDs, all situated in the same row (to limit inter-row replication traffic) but separated into different cabinets (to minimize exposure to a power circuit or edge switch failure). The cluster map also includes a list of down or inactive devices and an epoch number, which is incremented each time the map changes. All OSD requests are tagged with the client's map epoch, such that all parties can agree on the current distribution of data. Incremental map updates are shared between cooperating OSDs, and piggyback on OSD replies if the client's map is out of date.

5.2 Replication

In contrast to systems like Lustre [4], which assume one can construct sufficiently reliable OSDs using mechanisms like RAID or fail-over on a SAN, we assume that in a petabyte or exabyte system failure will be the norm rather than the exception, and at any point in time several OSDs are likely to be inoperable. To maintain system availability and ensure data safety in a scalable fashion, RADOS manages its own replication of data using a variant of primary-copy replication [2], while taking steps to minimize the impact on performance.

Data is replicated in terms of placement groups, each of which is mapped to an ordered list of n OSDs (for n-way replication). Clients send all writes to the first non-failed OSD in an object's PG (the primary), which assigns a new version number for the object and PG and forwards the write to any additional replica OSDs. After each replica has applied the update and responded to the primary, the primary applies the update locally and the write is acknowledged to the client. Reads are directed at the primary. This approach spares the client of any of the complexity surrounding synchronization or serialization between replicas, which can be onerous in the presence of other writers or failure recovery. It also shifts the bandwidth consumed by replication from the client to the OSD cluster's internal network, where we expect greater resources to be available. Intervening replica OSD failures are ignored, as any subsequent recovery (see Section 5.5) will reliably restore replica consistency.

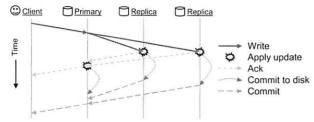


Figure 4: RADOS responds with an *ack* after the write has been applied to the buffer caches on all OSDs replicating the object. Only after it has been safely committed to disk is a final *commit* notification sent to the client.

5.3 Data Safety

In distributed storage systems, there are essentially two reasons why data is written to shared storage. First, clients are interested in making their updates visible to other clients. This should be quick: writes should be visible as soon as possible, particularly when multiple writers or mixed readers and writers force clients to operate synchronously. Second, clients are interested in knowing definitively that the data they've written is safely replicated, on disk, and will survive power or other failures. RADOS disassociates synchronization from safety when acknowledging updates, allowing Ceph to realize both low-latency updates for efficient application synchronization and well-defined data safety semantics.

Figure 4 illustrates the messages sent during an object write. The primary forwards the update to replicas, and replies with an *ack* after it is applied to all OSDs' in-memory buffer caches, allowing synchronous POSIX calls on the client to return. A final *commit* is sent (perhaps many seconds later) when data is safely committed to disk. We send the *ack* to the client only after the update is fully replicated to seamlessly tolerate the failure of any single OSD, even though this increases client latency. By default, clients also buffer writes until they commit to avoid data loss in the event of a simultaneous power loss to all OSDs in the placement group. When recovering in such cases, RADOS allows the replay of previously acknowledged (and thus ordered) updates for a fixed interval before new updates are accepted.

5.4 Failure Detection

Timely failure detection is critical to maintaining data safety, but can become difficult as a cluster scales to many thousands of devices. For certain failures, such as disk errors or corrupted data, OSDs can self-report. Failures that make an OSD unreachable on the network, however, require active monitoring, which RADOS distributes by having each OSD monitor those peers with which it shares PGs. In most cases, existing replication traffic serves as a passive confirmation of liveness, with no additional communication overhead. If an OSD has not heard from a peer recently, an explicit ping is sent.

RADOS considers two dimensions of OSD liveness: whether the OSD is reachable, and whether it is assigned data by CRUSH. An unresponsive OSD is initially marked *down*, and any primary responsibilities (update serialization, replication) temporarily pass to the next OSD in each of its placement groups. If the OSD does not quickly recover, it is marked *out* of the data distribution, and another OSD joins each PG to re-replicate its contents. Clients which have pending operations with a failed OSD simply resubmit to the new primary.

Because a wide variety of network anomalies may cause intermittent lapses in OSD connectivity, a small cluster of monitors collects failure reports and filters out transient or systemic problems (like a network partition) centrally. Monitors (which are only partially implemented) use elections, active peer monitoring, short-term leases, and two-phase commits to collectively provide consistent and available access to the cluster map. When the map is updated to reflect any failures or recoveries, affected OSDs are provided incremental map updates, which then spread throughout the cluster by piggybacking on existing inter-OSD communication. Distributed detection allows fast detection without unduly burdening monitors, while resolving the occurrence of inconsistency with centralized arbitration. Most importantly, RADOS avoids initiating widespread data re-replication due to systemic problems by marking OSDs down but not out (e.g., after a power loss to half of all OSDs).

5.5 Recovery and Cluster Updates

The OSD cluster map will change due to OSD failures, recoveries, and explicit cluster changes such as the deployment of new storage. Ceph handles all such changes in the same way. To facilitate fast recovery, OSDs maintain a version number for each object and a log of recent changes (names and versions of updated or deleted objects) for each PG (similar to the replication logs in Harp [14]).

When an active OSD receives an updated cluster map, it iterates over all locally stored placement groups and calculates the CRUSH mapping to determine which ones it is responsible for, either as a primary or replica. If a PG's membership has changed, or if the OSD has just booted, the OSD must peer with the PG's other OSDs. For replicated PGs, the OSD provides the primary with its current PG version number. If the OSD is the primary for the PG, it collects current (and former) replicas' PG versions. If the primary lacks the most recent PG state, it retrieves the log of recent PG changes (or a complete content summary, if needed) from current or prior OSDs in the PG in order to determine the correct (most recent) PG contents. The primary then sends each replica an incremental log update (or complete content summary, if needed), such that all parties know what the PG contents

should be, even if their locally stored object set may not match. Only after the primary determines the correct PG state and shares it with any replicas is I/O to objects in the PG permitted. OSDs are then independently responsible for retrieving missing or outdated objects from their peers. If an OSD receives a request for a stale or missing object, it delays processing and moves that object to the front of the recovery queue.

For example, suppose osd1 crashes and is marked down, and osd2 takes over as primary for pgA. If osd1 recovers, it will request the latest map on boot, and a monitor will mark it as up. When osd2 receives the resulting map update, it will realize it is no longer primary for pgA and send the pgA version number to osd1. osd1 will retrieve recent pgA log entries from osd2, tell osd2 its contents are current, and then begin processing requests while any updated objects are recovered in the background.

Because failure recovery is driven entirely by individual OSDs, each PG affected by a failed OSD will recover in parallel to (very likely) different replacement OSDs. This approach, based on the Fast Recovery Mechanism (FaRM) [37], decreases recovery times and improves overall data safety.

5.6 Object Storage with EBOFS

Although a variety of distributed file systems use local file systems like ext3 to manage low-level storage [4, 12], we found their interface and performance to be poorly suited for object workloads [27]. The existing kernel interface limits our ability to understand when object updates are safely committed on disk. Synchronous writes or journaling provide the desired safety, but only with a heavy latency and performance penalty. More importantly, the POSIX interface fails to support atomic data and metadata (*e. g.*, attribute) update transactions, which are important for maintaining RADOS consistency.

Instead, each Ceph OSD manages its local object storage with EBOFS, an Extent and B-tree based Object File System. Implementing EBOFS entirely in user space and interacting directly with a raw block device allows us to define our own low-level object storage interface and update semantics, which separate update serialization (for synchronization) from on-disk commits (for safety). EBOFS supports atomic transactions (*e. g.*, writes and attribute updates on multiple objects), and update functions return when the in-memory caches are updated, while providing asynchronous notification of commits.

A user space approach, aside from providing greater flexibility and easier implementation, also avoids cumbersome interaction with the Linux VFS and page cache, both of which were designed for a different interface and workload. While most kernel file systems lazily flush updates to disk after some time interval, EBOFS aggres-

sively schedules disk writes, and opts instead to cancel pending I/O operations when subsequent updates render them superfluous. This provides our low-level disk scheduler with longer I/O queues and a corresponding increase in scheduling efficiency. A user-space scheduler also makes it easier to eventually prioritize workloads (*e. g.*, client I/O versus recovery) or provide quality of service guarantees [36].

Central to the EBOFS design is a robust, flexible, and fully integrated B-tree service that is used to locate objects on disk, manage block allocation, and index collections (placement groups). Block allocation is conducted in terms of extents-start and length pairs-instead of block lists, keeping metadata compact. Free block extents on disk are binned by size and sorted by location, allowing EBOFS to quickly locate free space near the write position or related data on disk, while also limiting long-term fragmentation. With the exception of perobject block allocation information, all metadata is kept in memory for performance and simplicity (it is quite small, even for large volumes). Finally, EBOFS aggressively performs copy-on-write: with the exception of superblock updates, data is always written to unallocated regions of disk.

6 Performance and Scalability Evaluation

We evaluate our prototype under a range of microbenchmarks to demonstrate its performance, reliability, and scalability. In all tests, clients, OSDs, and MDSs are user processes running on a dual-processor Linux cluster with SCSI disks and communicating using TCP. In general, each OSD or MDS runs on its own host, while tens or hundreds of client instances may share the same host while generating workload.

6.1 Data Performance

EBOFS provides superior performance and safety semantics, while the balanced distribution of data generated by CRUSH and the delegation of replication and failure recovery allow aggregate I/O performance to scale with the size of the OSD cluster.

6.1.1 OSD Throughput

We begin by measuring the I/O performance of a 14-node cluster of OSDs. Figure 5 shows per-OSD throughput (y) with varying write sizes (x) and replication. Workload is generated by 400 clients on 20 additional nodes. Performance is ultimately limited by the raw disk bandwidth (around 58 MB/sec), shown by the horizontal line. Replication doubles or triples disk I/O, reducing client data rates accordingly when the number of OSDs is fixed.

Figure 6 compares the performance of EBOFS to that of general-purpose file systems (ext3, ReiserFS, XFS) in handling a Ceph workload. Clients synchronously

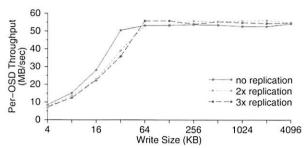


Figure 5: Per-OSD write performance. The horizontal line indicates the upper limit imposed by the physical disk. Replication has minimal impact on OSD throughput, although if the number of OSDs is fixed, n-way replication reduces total *effective* throughput by a factor of n because replicated data must be written to n OSDs.

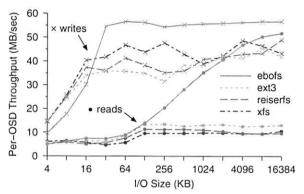


Figure 6: Performance of EBOFS compared to generalpurpose file systems. Although small writes suffer from coarse locking in our prototype, EBOFS nearly saturates the disk for writes larger than 32 KB. Since EBOFS lays out data in large extents when it is written in large increments, it has significantly better read performance.

write out large files, striped over 16 MB objects, and read them back again. Although small read and write performance in EBOFS suffers from coarse threading and locking, EBOFS very nearly saturates the available disk bandwidth for writes sizes larger than 32 KB, and significantly outperforms the others for read workloads because data is laid out in extents on disk that match the write sizes—even when they are very large. Performance was measured using a fresh file system. Experience with an earlier EBOFS design suggests it will experience significantly lower fragmentation than ext3, but we have not yet evaluated the current implementation on an aged file system. In any case, we expect the performance of EBOFS after aging to be no worse than the others.

6.1.2 Write Latency

Figure 7 shows the synchronous write latency (y) for a single writer with varying write sizes (x) and replica-

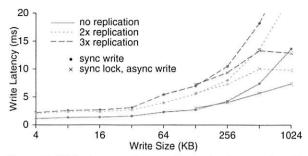


Figure 7: Write latency for varying write sizes and replication. More than two replicas incurs minimal additional cost for small writes because replicated updates occur concurrently. For large synchronous writes, transmission times dominate. Clients partially mask that latency for writes over 128 KB by acquiring exclusive locks and asynchronously flushing the data.

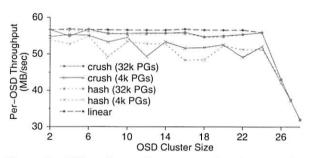


Figure 8: OSD write performance scales linearly with the size of the OSD cluster until the switch is saturated at 24 OSDs. CRUSH and hash performance improves when more PGs lower variance in OSD utilization.

tion. Because the primary OSD simultaneously retransmits updates to all replicas, small writes incur a minimal latency increase for more than two replicas. For larger writes, the cost of retransmission dominates; 1 MB writes (not shown) take 13 ms for one replica, and 2.5 times longer (33 ms) for three. Ceph clients partially mask this latency for synchronous writes over 128 KB by acquiring exclusive locks and then asynchronously flushing the data to disk. Alternatively, write-sharing applications can opt to use O_LAZY. With consistency thus relaxed, clients can buffer small writes and submit only large, asynchronous writes to OSDs; the only latency seen by applications will be due to clients which fill their caches waiting for data to flush to disk.

6.1.3 Data Distribution and Scalability

Ceph's data performance scales nearly linearly in the number of OSDs. CRUSH distributes data pseudorandomly such that OSD utilizations can be accurately modeled by a binomial or normal distribution—what one expects from a perfectly random process [29]. Vari-

ance in utilizations decreases as the number of groups increases: for 100 placement groups per OSD the standard deviation is 10%; for 1000 groups it is 3%. Figure 8 shows per-OSD write throughput as the cluster scales using CRUSH, a simple hash function, and a linear striping strategy to distribute data in 4096 or 32768 PGs among available OSDs. Linear striping balances load perfectly for maximum throughput to provide a benchmark for comparison, but like a simple hash function, it fails to cope with device failures or other OSD cluster changes. Because data placement with CRUSH or a hash is stochastic, throughputs are lower with fewer PGs: greater variance in OSD utilizations causes request queue lengths to drift apart under our entangled client workload. Because devices can become overfilled or overutilized with small probability, dragging down performance, CRUSH can correct such situations by offloading any fraction of the allocation for OSDs specially marked in the cluster map. Unlike the hash and linear strategies, CRUSH also minimizes data migration under cluster expansion while maintaining a balanced distribution. CRUSH calculations are $O(\log n)$ (for a cluster of n OSDs) and take only tens of microseconds, allowing clusters to grow to hundreds of thousands of OSDs.

6.2 Metadata Performance

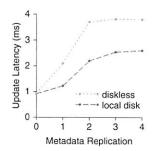
Ceph's MDS cluster offers enhanced POSIX semantics with excellent scalability. We measure performance via a partial workload lacking any data I/O; OSDs in these experiments are used solely for metadata storage.

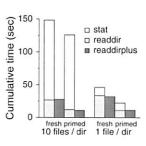
6.2.1 Metadata Update Latency

We first consider the latency associated with metadata updates (e. g., mknod or mkdir). A single client creates a series of files and directories which the MDS must synchronously journal to a cluster of OSDs for safety. We consider both a diskless MDS, where all metadata is stored in a shared OSD cluster, and one which also has a local disk serving as the primary OSD for its journal. Figure 9(a) shows the latency (y) associated with metadata updates in both cases with varying metadata replication (x) (where zero corresponds to no journaling at all). Journal entries are first written to the primary OSD and then replicated to any additional OSDs. With a local disk, the initial hop from the MDS to the (local) primary OSD takes minimal time, allowing update latencies for $2\times$ replication similar to $1\times$ in the diskless model. In both cases, more than two replicas incurs little additional latency because replicas update in parallel.

6.2.2 Metadata Read Latency

The behavior of metadata reads (e. g., readdir, stat, open) is more complex. Figure 9(b) shows cumulative time (y) consumed by a client walking 10,000 nested directories with a readdir in each directory and a stat on each file. A





(a) Metadata update latency for an MDS with and without a local disk. Zero corresponds to no journaling.

(b) Cumulative time consumed during a file system walk.

Figure 9: Using a local disk lowers the write latency by avoiding the initial network round-trip. Reads benefit from caching, while *readdirplus* or relaxed consistency eliminate MDS interaction for *stats* following *readdir*.

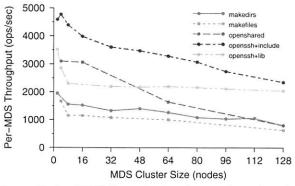


Figure 10: Per-MDS throughput under a variety of work-loads and cluster sizes. As the cluster grows to 128 nodes, efficiency drops no more than 50% below perfect linear (horizontal) scaling for most workloads, allowing vastly improved performance over existing systems.

primed MDS cache reduces *readdir* times. Subsequent *stats* are not affected, because inode contents are embedded in directories, allowing the full directory contents to be fetched into the MDS cache with a single OSD access. Ordinarily, cumulative *stat* times would dominate for larger directories. Subsequent MDS interaction can be eliminated by using *readdirplus*, which explicitly bundles *stat* and *readdir* results in a single operation, or by relaxing POSIX to allow *stats* immediately following a *readdir* to be served from client caches (the default).

6.2.3 Metadata Scaling

We evaluate metadata scalability using a 430 node partition of the alc Linux cluster at Lawrence Livermore National Laboratory (LLNL). Figure 10 shows per-MDS throughput (y) as a function of MDS cluster size (x), such that a horizontal line represents perfect linear scaling. In the *makedirs* workload, each client creates a tree

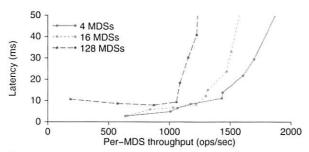


Figure 11: Average latency versus per-MDS throughput for different cluster sizes (*makedirs* workload).

of nested directories four levels deep, with ten files and subdirectories in each directory. Average MDS throughput drops from 2000 ops per MDS per second with a small cluster, to about 1000 ops per MDS per second (50% efficiency) with 128 MDSs (over 100,000 ops/sec total). In the makefiles workload, each client creates thousands of files in the same directory. When the high write levels are detected, Ceph hashes the shared directory and relaxes the directory's mtime coherence to distribute the workload across all MDS nodes. The openshared workload demonstrates read sharing by having each client repeatedly open and close ten shared files. In the openssh workloads, each client replays a captured file system trace of a compilation in a private directory. One variant uses a shared /lib for moderate sharing, while the other shares /usr/include, which is very heavily read. The openshared and openssh+include workloads have the heaviest read sharing and show the worst scaling behavior, we believe due to poor replica selection by clients. openssh+lib scales better than the trivially separable makedirs because it contains relatively few metadata modifications and little sharing. Although we believe that contention in the network or threading in our messaging layer further lowered performance for larger MDS clusters, our limited time with dedicated access to the large cluster prevented a more thorough investigation.

Figure 11 plots latency (y) versus per-MDS throughput (x) for a 4-, 16-, and 64-node MDS cluster under the *makedirs* workload. Larger clusters have imperfect load distributions, resulting in lower average per-MDS throughput (but, of course, much higher total throughput) and slightly higher latencies.

Despite imperfect linear scaling, a 128-node MDS cluster running our prototype can service more than a quarter million metadata operations per second (128 nodes at 2000 ops/sec). Because metadata transactions are independent of data I/O and metadata size is independent of file size, this corresponds to installations with potentially many hundreds of petabytes of storage or more, depending on average file size. For example, scientific applications creating checkpoints on LLNL's Bluegene/L

might involve 64 thousand nodes with two processors each writing to separate files in the same directory (as in the makefiles workload). While the current storage system peaks at 6,000 metadata ops/sec and would take minutes to complete each checkpoint, a 128-node Ceph MDS cluster could finish in two seconds. If each file were only 10 MB (quite small by HPC standards) and OSDs sustain 50 MB/sec, such a cluster could write 1.25 TB/sec, saturating at least 25,000 OSDs (50,000 with replication). 250 GB OSDs would put such a system at more than six petabytes. More importantly, Ceph's dynamic metadata distribution allows an MDS cluster (of any size) to reallocate resources based on the current workload, even when all clients access metadata previously assigned to a single MDS, making it significantly more versatile and adaptable than any static partitioning strategy.

7 Experiences

We were pleasantly surprised by the extent to which replacing file allocation metadata with a distribution function became a simplifying force in our design. Although this placed greater demands on the function itself, once we realized exactly what those requirements were, CRUSH was able to deliver the necessary scalability, flexibility, and reliability. This vastly simplified our metadata workload while providing both clients and OSDs with complete and independent knowledge of the data distribution. The latter enabled us to delegate responsibility for data replication, migration, failure detection, and recovery to OSDs, distributing these mechanisms in a way that effectively leveraged their bundled CPU and memory. RADOS has also opened the door to a range of future enhancements that elegantly map onto our OSD model, such as bit error detection (as in the Google File System [7]) and dynamic replication of data based on workload (similar to AutoRAID [34]).

Although it was tempting to use existing kernel file systems for local object storage (as many other systems have done [4, 7, 9]), we recognized early on that a file system tailored for object workloads could offer better performance [27]. What we did not anticipate was the disparity between the existing file system interface and our requirements, which became evident while developing the RADOS replication and reliability mechanisms. EBOFS was surprisingly quick to develop in user-space, offered very satisfying performance, and exposed an interface perfectly suited to our requirements.

One of the largest lessons in Ceph was the importance of the MDS load balancer to overall scalability, and the complexity of choosing what metadata to migrate where and when. Although in principle our design and goals seem quite simple, the reality of distributing an evolving workload over a hundred MDSs highlighted additional subtleties. Most notably, MDS performance has

a wide range of performance bounds, including CPU, memory (and cache efficiency), and network or I/O limitations, any of which may limit performance at any point in time. Furthermore, it is difficult to quantitatively capture the balance between total throughput and fairness; under certain circumstances unbalanced metadata distributions can increase overall throughput [30].

Implementation of the client interface posed a greater challenge than anticipated. Although the use of FUSE vastly simplified implementation by avoiding the kernel, it introduced its own set of idiosyncrasies. DIRECT_IO bypassed kernel page cache but didn't support *mmap*, forcing us to modify FUSE to invalidate clean pages as a workaround. FUSE's insistence on performing its own security checks results in copious *getattrs* (*stats*) for even simple application calls. Finally, page-based I/O between kernel and user space limits overall I/O rates. Although linking directly to the client avoids FUSE issues, overloading system calls in user space introduces a new set of issues (most of which we have yet to fully examine), making an in-kernel client module inevitable.

8 Related Work

High-performance scalable file systems have long been a goal of the HPC community, which tends to place a heavy load on the file system [18, 27]. Although many file systems attempt to meet this need, they do not provide the same level of scalability that Ceph does. Largescale systems like OceanStore [11] and Farsite [1] are designed to provide petabytes of highly reliable storage, and can provide simultaneous access to thousands of separate files to thousands of clients, but cannot provide high-performance access to a small set of files by tens of thousands of cooperating clients due to bottlenecks in subsystems such as name lookup. Conversely, parallel file and storage systems such as Vesta [6], Galley [17], PVFS [12], and Swift [5] have extensive support for striping data across multiple disks to achieve very high transfer rates, but lack strong support for scalable metadata access or robust data distribution for high reliability. For example, Vesta permits applications to lay their data out on disk, and allows independent access to file data on each disk without reference to shared metadata. However, like many other parallel file systems, Vesta does not provide scalable support for metadata lookup. As a result, these file systems typically provide poor performance on workloads that access many small files or require many metadata operations. They also typically suffer from block allocation issues: blocks are either allocated centrally or via a lock-based mechanism, preventing them from scaling well for write requests from thousands of clients to thousands of disks. GPFS [24] and StorageTank [16] partially decouple metadata and data management, but are limited by their use of block-based disks and their metadata distribution architecture.

Grid-based file systems such as LegionFS [33] are designed to coordinate wide-area access and are not optimized for high performance in the local file system. Similarly, the Google File System [7] is optimized for very large files and a workload consisting largely of reads and file appends. Like Sorrento [26], it targets a narrow class of applications with non-POSIX semantics.

Recently, many file systems and platforms, including Federated Array of Bricks (FAB) [23] and pNFS [9] have been designed around network attached storage [8]. Lustre [4], the Panasas file system [32], zFS [21], Sorrento, and Kybos [35] are based on the object-based storage paradigm [3] and most closely resemble Ceph. However, none of these systems has the combination of scalable and adaptable metadata management, reliability and fault tolerance that Ceph provides. Lustre and Panasas in particular fail to delegate responsibility to OSDs, and have limited support for efficient distributed metadata management, limiting their scalability and performance. Further, with the exception of Sorrento's use of consistent hashing [10], all of these systems use explicit allocation maps to specify where objects are stored, and have limited support for rebalancing when new storage is deployed. This can lead to load asymmetries and poor resource utilization, while Sorrento's hashed distribution lacks CRUSH's support for efficient data migration, device weighting, and failure domains.

9 Future Work

Some core Ceph elements have not yet been implemented, including MDS failure recovery and several POSIX calls. Two security architecture and protocol variants are under consideration, but neither have yet been implemented [13, 19]. We also plan on investigating the practicality of client callbacks on namespace to inode translation metadata. For static regions of the file system, this could allow opens (for read) to occur without MDS interaction. Several other MDS enhancements are planned, including the ability to create snapshots of arbitrary subtrees of the directory hierarchy [28].

Although Ceph dynamically replicates metadata when flash crowds access single directories or files, the same is not yet true of file data. We plan to allow OSDs to dynamically adjust the level of replication for individual objects based on workload, and to distribute read traffic across multiple OSDs in the placement group. This will allow scalable access to small amounts of data, and may facilitate fine-grained OSD load balancing using a mechanism similar to D-SPTF [15].

Finally, we are working on developing a quality of service architecture to allow both aggregate classbased traffic prioritization and OSD-managed reservation based bandwidth and latency guarantees. In addition to supporting applications with QoS requirements, this will help balance RADOS replication and recovery operations with regular workload. A number of other EBOFS enhancements are planned, including improved allocation logic, data scouring, and checksums or other bit-error detection mechanisms to improve data safety.

10 Conclusions

Ceph addresses three critical challenges of storage systems—scalability, performance, and reliability—by occupying a unique point in the design space. By shedding design assumptions like allocation lists found in nearly all existing systems, we maximally separate data from metadata management, allowing them to scale independently. This separation relies on CRUSH, a data distribution function that generates a pseudo-random distribution, allowing clients to calculate object locations instead of looking them up. CRUSH enforces data replica separation across failure domains for improved data safety while efficiently coping with the inherently dynamic nature of large storage clusters, where devices failures, expansion and cluster restructuring are the norm.

RADOS leverages intelligent OSDs to manage data replication, failure detection and recovery, low-level disk allocation, scheduling, and data migration without encumbering any central server(s). Although objects can be considered files and stored in a general-purpose file system, EBOFS provides more appropriate semantics and superior performance by addressing the specific workloads and interface requirements present in Ceph.

Finally, Ceph's metadata management architecture addresses one of the most vexing problems in highly scalable storage—how to efficiently provide a single uniform directory hierarchy obeying POSIX semantics with performance that scales with the number of metadata servers. Ceph's dynamic subtree partitioning is a uniquely scalable approach, offering both efficiency and the ability to adapt to varying workloads.

Ceph is licensed under the LGPL and is available at http://ceph.sourceforge.net/.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48. Research was funded in part by the Lawrence Livermore, Los Alamos, and Sandia National Laboratories. We would like to thank Bill Loewe, Tyce McLarty, Terry Heidelberg, and everyone else at LLNL who talked to us about their storage trials and tribulations, and who helped facilitate our two days of dedicated access time on alc. We would also like to

thank IBM for donating the 32-node cluster that aided in much of the OSD performance testing, and the National Science Foundation, which paid for the switch upgrade. Chandu Thekkath (our shepherd), the anonymous reviewers, and Theodore Wong all provided valuable feedback, and we would also like to thank the students, faculty, and sponsors of the Storage Systems Research Center for their input and support.

References

- [1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.
- [2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the* 2nd International Conference on Software Engineering, pages 562–570. IEEE Computer Society Press, 1976.
- [3] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi. Towards an object store. In *Proceedings* of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies, pages 165–176, Apr. 2003.
- [4] P. J. Braam. The Lustre storage architecture. http://www.lustre.org/documentation.html, Cluster File Systems, Inc., Aug. 2004.
- [5] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, 1991.
- [6] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. ACM Transactions on Computer Systems, 14(3):225–264, 1996.
- [7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium* on *Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, Oct. 2003. ACM.
- [8] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 92–103, San Jose, CA, Oct. 1998.
- [9] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. Technical Report CITI-05-1, CITI, University of Michigan, Feb. 2005.
- [10] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In ACM Symposium on Theory of Computing, pages 654–663, May 1997.
- [11] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for

- global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASP-LOS)*, Cambridge, MA, Nov. 2000. ACM.
- [12] R. Latham, N. Miller, R. Ross, and P. Carns. A next-generation parallel file system for Linux clusters. *Linux-World*, pages 56–59, Jan. 2004.
- [13] A. Leung and E. L. Miller. Scalable security for large, high performance storage systems. In *Proceedings of the* 2006 ACM Workshop on Storage Security and Survivability. ACM, Oct. 2006.
- [14] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 226– 238. ACM, 1991.
- [15] C. R. Lumb, G. R. Ganger, and R. Golding. D-SPTF: Decentralized request distribution in brick-based storage systems. In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 37–47, Boston, MA, 2004.
- [16] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM Storage Tank—a heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2):250– 267, 2003.
- [17] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, 1996. ACM Press.
- [18] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, Oct. 1996.
- [19] C. A. Olson and E. L. Miller. Secure capabilities for a petabyte-scale object-based distributed file system. In Proceedings of the 2005 ACM Workshop on Storage Security and Survivability, Fairfax, VA, Nov. 2005.
- [20] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137–151, 1994.
- [21] O. Rodeh and A. Teperman. zFS—a scalable distributed file system using object disks. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 207–218, Apr. 2003.
- [22] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, San Diego, CA, June 2000. USENIX Association.
- [23] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of* the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 48–58, 2004.
- [24] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of* the 2002 Conference on File and Storage Technologies (FAST), pages 231–244. USENIX, Jan. 2002.

- [25] M. Szeredi. File System in User Space. http://fuse.sourceforge.net, 2006.
- [26] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang, and L. Chu. A self-organizing storage cluster for parallel data-intensive applications. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, Pittsburgh, PA, Nov. 2004.
- [27] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 139–152, College Park, MD, Apr. 2004.
- [28] S. A. Weil. Scalable archival data and metadata management in object-based file systems. Technical Report SSRC-04-01, University of California, Santa Cruz, May 2004.
- [29] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*, Tampa, FL, Nov. 2006, ACM.
- [30] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Con*ference on Supercomputing (SC '04), ACM, Nov. 2004.
- [31] B. Welch. POSIX IO extensions for HPC. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, Dec. 2005.
- [32] B. Welch and G. Gibson. Managing scalability in object storage systems for HPC Linux clusters. In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies, pages 433–445, Apr. 2004.
- [33] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw. LegionFS: A secure and scalable file system supporting cross-domain high-performance applications. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC '01)*, Denver, CO, 2001.
- [34] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 96–108, Copper Mountain, CO, 1995. ACM Press.
- [35] T. M. Wong, R. A. Golding, J. S. Glider, E. Borowsky, R. A. Becker-Szendy, C. Fleiner, D. R. Kenchammana-Hosekote, and O. A. Zaki. Kybos: self-management for distributed brick-base storage. Research Report RJ 10356, IBM Almaden Research Center, Aug. 2005.
- [36] J. C. Wu and S. A. Brandt. The design and implementation of AQuA: an adaptive quality of service aware object-based storage device. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 209–218, College Park, MD, May 2006.
- [37] Q. Xin, E. L. Miller, and T. J. E. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC), pages 172–181, Honolulu, HI, June 2004.

Distributed Directory Service in the Farsite File System

John R. Douceur and Jon Howell Microsoft Research, Redmond, WA 98052 {johndo, howell}@microsoft.com

Abstract

We present the design, implementation, and evaluation of a fully distributed directory service for Farsite, a logically centralized file system that is physically implemented on a loosely coupled network of desktop computers. Prior to this work, the Farsite system included distributed mechanisms for file content but centralized mechanisms for file metadata. Our distributed directory service introduces tree-structured file identifiers that support dynamically partitioning metadata at arbitrary granularity, recursive path leases for scalably maintaining name-space consistency, and a protocol for consistently performing operations on files managed by separate machines. It also mitigates metadata hotspots via file-field leases and the new mechanism of disjunctive leases. We experimentally show that Farsite can dynamically partition file-system metadata while maintaining full file-system semantics.

1 Introduction

Farsite [1] is a serverless, distributed file system that logically functions as a centralized file server but is physically distributed among a network of desktop workstations. Farsite provides the location-transparent file access and reliable data storage commonly provided by a central file server, without a central file server's infrastructure costs, susceptibility to geographically localized faults, and reliance on the error-freedom and trustworthiness of system administrators.

Farsite replaces the physical security of a locked room with the virtual security of cryptography and Byzantine-fault tolerance (BFT) [7]. It replaces high-reliability server hardware with distributed redundant storage, computation, and communication provided by the unused resources of existing desktop machines. And it replaces central system administrators with autonomic processes that transparently migrate replicas of file content and file-system metadata to adapt to variations in applied load, changes in machines' availability, and the arrival and departure of machines.

In 2002, we published a paper [1] describing a Farsite system that provides a Windows file-system interface via a file-system driver, that stores encrypted replicas of file content on remote machines, that autonomically rearranges these replicas to maintain file availability [11], that provides transparent access to remotely stored file content, that caches content on clients for efficiency, and that lazily propagates file updates to remote replicas. This implementation also includes a directory service that manages file-system metadata, issues leases to clients, and receives metadata updates from clients. The directory service tolerates malicious clients by validating metadata updates before applying them to the persistent metadata state. In addition, the directory service tolerates failures or malice in the machines on which it runs, by replicating its execution on a BFT group of machines [7].

However, because every machine in a BFT group redundantly executes the directory-service code, the group has no more throughput than a single machine does, irrespective of the group size. Since this directory service runs on a single BFT group, the rate at which the group can process file-system metadata governs the maximum scale of the system. By contrast, every other Farsite subsystem avoids scalability limitations either by implicit parallelism or by relaxed consistency that is tolerable because it is concealed by the directory service.

This paper presents a new design, implementation, and evaluation of a directory service that is scalable and seamlessly distributed. This work involves several novel techniques for partitioning file-system metadata, maintaining name-space consistency, coordinating multi-machine operations, and mitigating metadata hotspots. Experiments in a modest-sized configuration show that our techniques enable the system to sustain throughput that is proportional to system scale.

Our contributions include the following techniques:

- · tree-structured file identifiers
- multi-machine operation protocol
- recursive path leases
- file-field leases
- disjunctive leases

Our contributions also include the end result, namely:

 a file system that seamlessly distributes and dynamically repartitions metadata among multiple machines

Section 2 overviews the Farsite system. Section 3 describes the design decisions that drove the techniques detailed in Section 4 on metadata partitioning and Section 5 on hotspot mitigation. Section 6 reports on design lessons we learned, and Section 7 describes our implementation. Section 8 experimentally evaluates the system. Sections 9 and 10 describe future work and related work. Section 11 summarizes and concludes.

2 Farsite overview

This section briefly summarizes the goals, assumptions, and basic system design of Farsite. Many more details can be found in our 2002 paper [1].

2.1 Goals

Farsite is Windows-based file system intended to replace the central file servers in a large corporation or university, which can serve as many as $\sim 10^5$ desktop clients [4]. Of Farsite's many design goals, the key goal for the present work is minimizing human system administration, which is not only a significant cost for large server systems but also a major cause of system failure [26]. In this paper, we address the specific subgoal of automated load balancing.

2.2 Assumptions

Farsite is intended for wired campus-area networks, wherein all machines can communicate with each other within a small number of milliseconds, and network topology can be mostly ignored. Farsite expects that machines may fail and that network connections may be flaky, but it is not designed to gracefully handle long-term disconnections.

Farsite's intended workload is that of normal desktop systems, which exhibit high access locality, a low rate of updates that survive subsequent overwrites, and infrequent, small-scale read/write sharing [20, 41]. It is not intended for high-performance parallel I/O.

2.3 Basic system design

This section briefly overviews the aspects of the Farsite system relevant to the metadata service, completely ignoring issues relating to file content.

Each machine functions in two roles, a *client* that performs operations for its local user and a *server* that provides directory service to clients. To perform an operation, a client obtains a copy of the relevant metadata from the server, along with a lease [15] over the metadata. For the duration of the lease, the client has an authoritative value for the metadata. If the lease is a write lease, then the server temporarily gives up authority over the metadata, because the lease permits the client to modify the metadata.

After the client performs the metadata operation, it lazily propagates its metadata updates to the server. To make it easier for servers to validate a client's updates, the updates are described as operations rather than as deltas to metadata state.

When the load on a server becomes excessive, the server selects another machine in the system and *delegates* a portion of its metadata to this machine.

From the perspective of the directory service, there is little distinction between directories and data files.

other than directories have no content and data files may not have children. For simplicity, we refer to them both as "files" except when clarity dictates otherwise.

3 Directory service design decisions

This section motivates our design decisions and describes why we did not follow through with many of the design ideas enumerated in our 2002 paper.

3.1 Fully functional rename

A fundamental decision was to support a fully functional rename operation. Decades of experience by Unix and Windows users have shown that fully functional rename is part of what makes a hierarchical file system a valuable tool for organizing data. In fact, rename is the only operation that makes non-trivial use of the name space's hierarchy, by atomically changing the paths of every file in a subtree while preserving open handles. Without rename, the file-system structure is only slightly more involved than a flat name space wherein the path separator ('/' in Unix, '\' in Windows) is just another character. It is slightly more involved because, for example, one can create a file only if its parent exists, so in the flattened name space, one could create a new name only if a constrained prefix of the name already exists.

Some prior distributed file systems have divided the name space into user-visible partitions and disallowed renames across partitions; examples include volumes in AFS [19], shares in Dfs [25], and domains in Sprite [27]. Anecdotal evidence suggests this is quite tolerable as long as the system administrator exercises care when selecting files for each partition: "[T]hey need to be closely related, in order for rename to be useful, and there should be enough of them so that the restriction on rename is not perceived by users as a problem [34]." However, in keeping with our goal of minimizing system administration, we shied away from user-visible partitions, which demand such careful organization [6]. Instead, we designed our metadata service to present a semantically seamless name space.

3.2 Name-space consistency

Given a fully functional rename operation, name-space consistency is necessary to avoid permanently disconnecting parts of the file system, as a simplified example illustrates. Fig. 1a shows a file-system tree in which every file is managed by a different server. Client *X* renames file C to be a child of file G, as shown in Fig. 1b, and Client *Y* independently renames file F to be a child of file D, as shown in Fig. 1c. No single server is directly involved in both rename operations, and each independent rename is legal. Yet, if both renames were allowed to proceed, the result would be an orphaned loop, as shown in Fig. 1d.

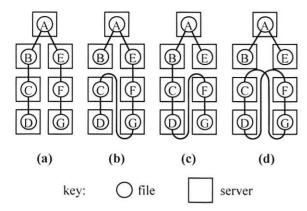


Fig. 1: Orphaned Loop from Two Renames

Once we had a mechanism for scalable name-space consistency, it seemed reasonable to use this mechanism to provide a consistent name space for all path-based operations, not merely for renames.

3.3 Long-term client disconnection

Although Farsite is not intended to gracefully support long-term client disconnections, we still need to decide what to do when such disconnections occur. Since we employ a lease-based framework, we attach expiration times – typically a few hours – to all leases issued by servers. When a lease expires, the server reclaims its authority over the leased metadata.

If a client had performed operations under the authority of this lease, these operations are effectively undone when the lease expires. Farsite thus permits lease expiration to cause metadata loss, not merely file content loss as in previous distributed file systems. However, these situations are not as radically different as they may first appear, because the user-visible semantics depend on how applications use files.

For example, Microsoft's email program Outlook [28] stores all of its data, including application-level metadata, in a single file. By contrast, maildir [40] uses the file system's name space to maintain email folder metadata. Under Outlook, content-lease expiration can cause email folder metadata updates to get lost; whereas under maildir, expiring a file-system metadata lease can cause the same behavior.

3.4 Prior design ideas

Our 2002 paper [1] envisioned a distributed directory service, for which it presented several ideas that seemed reasonable in abstract, but which proved problematic as we developed a detailed design.

Our intent had been to partition file metadata among servers according to file path names. Each client would maintain a cache of mappings from path names to their managing servers, similar to a Sprite prefix table [43]. The client could verify the authority of the

server over the path name by evaluating a chain of delegation certificates extending back to the root server. To diffuse metadata hotspots, servers would issue stale snapshots instead of leases when the client load got too high, and servers would lazily propagate the result of rename operations throughout the name space.

Partitioning by path name complicates renames across partitions, as detailed in the next section. In the absence of path-name delegation, name-based prefix tables are inappropriate. Similarly, if partitioning is not based on names, consistently resolving a path name requires access to metadata from all files along a path, so delegation certificates are unhelpful for scalability. Stale snapshots and lazy rename propagation allow the name space to become inconsistent, which can cause orphaned loops, as described above in Section 3.2. For these reasons, we abandoned these design ideas.

4 Metadata partitioning

In building a distributed metadata service, the most fundamental decision is how to partition the metadata among servers. One approach is to partition by file path name, as in Sprite [27] and the precursor to AFS [32], wherein each server manages files in a designated region of name space. However, this implies that when a file is renamed, either the file's metadata must migrate to the server that manages the destination name, or authority over the destination name must be delegated to the server that manages the file's metadata. The former approach – migration without delegation – is insufficient, because if a large subtree is being renamed, it may be not be manageable by a single server. The latter approach may be plausible, but coupling delegation to rename restricts the system's ability to use delegation for its primary purpose of load balancing. It also precludes clean software lavering. wherein a file's semantic attributes, including its name, are built on an abstraction that hides a file's mobility.

The problems with partitioning by path name arise from a name's mutability. We avoid these problems by partitioning according to file identifiers, which are not mutable. Our file identifiers have a tree structure, which supports arbitrarily fine-grained partitioning (§ 4.1). This partitioning is not user-visible, because operations can span multiple servers (§ 4.2). Despite partitioning, the name space is kept consistent – in a scalable fashion – by means of recursive path leases (§ 4.3).

4.1 Tree-structured file identifiers

4.1.1 Previous approach: flat file identifiers

Partitioning by file identifier is not original to Farsite; it is also the approach taken by AFS [19] and xFS [2]. All three systems need to efficiently store and retrieve information on which server manages each identifier.

AFS addresses this problem with *volumes* [34], and xFS addresses it with a similar unnamed abstraction.

An AFS file identifier is the concatenation of two components: The "volume identifier," which indicates the volume to which the file belongs, and the "file number," which uniquely identifies a file within the volume. All files in a volume reside on the same server. Volumes can be dynamically relocated among servers, but files cannot be dynamically repartitioned among volumes without reassigning file identifiers. An xFS "file index number" is similar; we defer more detailed discussion to Section 10 on related work.

4.1.2 File identifier design issues

A main design goal for Farsite's file identifiers was to enable metadata repartitioning without reassigning file identifiers. Specifically, we considered four issues:

- To maximize delegation-policy freedom, regions of identifier space should be partitionable with arbitrary granularity.
- To permit growth, each server should manage an unbounded region of file-identifier space.
- 3. For efficiency, file identifiers should have a compact representation.
- Also for efficiency, the (dynamic) mapping from file identifiers to servers should be stored in a time- and space-efficient structure.

4.1.3 Abstract structure

Abstractly, a file identifier is a sequence of positive integers, wherein the null sequence identifies the file-system root. Each server manages identifiers beginning with a specified prefix, except for those it has explicitly delegated away to other servers. Fig. 2 shows a system of six servers, A - F. The root server, A, manages all files except those whose identifiers are prefixed by $\langle 1 \rangle$, $\langle 3 \rangle$, or $\langle 4 \rangle$; server B manages all files with identifiers prefixed by $\langle 1 \rangle$ but not by $\langle 1.3 \rangle$; and so forth. The file identifier space is thus a tree, and servers manage subtrees of this space.

At any moment, some portion of each file identifier determines which server manages the file; however, the size of this portion is not fixed over time, unlike AFS

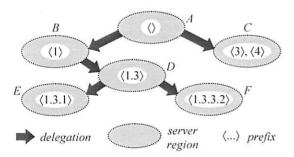


Fig. 2: Delegation of File Identifier Space

file identifiers. For example, in the partitioning of Fig. 2, a file with identifier $\langle 1.3.3.1 \rangle$ is managed by server D, because the longest delegated prefix of $\langle 1.3.3.1 \rangle$ is $\langle 1.3 \rangle$. If server D later delegates prefix $\langle 1.3.3 \rangle$, the file's managing server will be determined by the first three integers in the file identifier, namely $\langle 1.3.3 \rangle$. This arbitrary partitioning granularity addresses issue I above.

Because file identifiers have unbounded sequence length, each server manages an infinite set of file identifiers. For example, server D can create a new file identifier by appending any integer sequence to prefix $\langle 1.3 \rangle$, as long as the resulting identifier is not prefixed by $\langle 1.3.1 \rangle$ or $\langle 1.3.3.2 \rangle$. This addresses issue 2.

4.1.4 Implementation

A file identifier's integer sequence is encoded into a bit string using a variant of Elias γ' coding [13]. Because of the rule by which new file identifiers are assigned (§ 4.1.5), the frequency distribution of integers within file identifiers nearly matches the frequency distribution of files per directory. We chose γ' coding because it is optimally compact for the distribution we measured in a large-scale study [9], thereby addressing issue 3.

Variable-length identifiers may be unusual, but they are not complicated in practice. Our code includes a file-identifier class that stores small identifiers in an immediate variable and spills large identifiers onto the heap. Class encapsulation hides the length variability from all other parts of the system.

To store information about which servers manage which regions of file-identifier space, clients use a *file map*, which is similar to a Sprite prefix table [43], except it operates on prefixes of file identifiers rather than of path names. The file map is stored in an index structure adapted from Lampson et al.'s system for performing longest-matching-prefix search via binary search [23]. With respect to the count of mapped prefixes, the storage cost is linear, and the lookup time is logarithmic, thereby addressing issue 4.

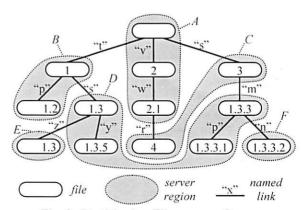


Fig. 3: Partitioning Files among Servers

4.1.5 Creating and renaming files

Each newly created file is assigned an identifier formed from its parent's identifier suffixed by an additional integer. For example, in Fig. 3, the file named "y" under file $\langle 1.3 \rangle$ was created with identifier $\langle 1.3.5 \rangle$. This rule tends to keep files that are close in the name space also close in the identifier space, so partitioning the latter produces few cuts in the former. This minimizes the work of path resolution, which is proportional to the number of server regions along a path.

Renames can disrupt the alignment of the name and identifier spaces. For example, Fig. 3 shows file $\langle 1.3.3 \rangle$ having been renamed from its original name under file $\langle 1.3 \rangle$ to name "m" under file $\langle 3 \rangle$. Unless renames occur excessively, there will still be enough alignment to keep the path-resolution process efficient.

4.2 Multi-server operations

The Windows rename operation atomically updates metadata for three files: the source directory, the destination directory, and the file being renamed. (Our protocol does not support POSIX-style rename, which enables a fourth file to be overwritten by the rename, although our protocol could possibly be extended to handle this case.) These files may be managed by three separate servers, so the client must obtain leases from all three servers before performing the rename. Before the client returns its leases, thus transferring authority over the metadata back to the managing servers, its metadata update must be validated by all three servers with logical atomicity.

The servers achieve this atomicity with two-phase locking: The server that manages the destination directory acts as the *leader*, and the other two servers act as *followers*. Each follower validates its part of the rename, *locks* the relevant metadata fields, and notifies the leader. The leader decides whether the update is valid and tells the followers to abort or commit their updates, either of which unlocks the field. While a field is locked, the server will not issue a lease on the field.

Since a follower that starts a multi-server operation is obligated to commit if the leader commits, a follower cannot unlock a field on its own, even to timeout a spurious update from a faulty client. Instead, the leader centrally handles timeouts by setting a timer for each notification it receives from a follower.

The followers must guarantee that the rename's preconditions still hold by the time the commit is received, which they can do because their preconditions are local and therefore lockable. The leader, which manages the destination server, can afford to check the one non-local condition, namely that the file being moved is not an ancestor of the destination. This check is facilitated by means of a path lease, as described in the following section.

4.3 Recursive path leases

As described in Section 3.2, servers require a consistent view of the name space to ensure that a rename operation does not produce an orphaned loop. More generally, Farsite provides name-space consistency for all path-based operations. The naive way to do this is for the servers managing all files along a path to issue individual leases on their files' children; however, this approach has poor scaling properties. In particular, it makes the root server responsible for providing leases to all interested parties in the system.

Our solution to this problem is the mechanism of recursive path leases. A file's path lease is a read-only lease on the chain of file identifiers of all files on the path from the file-system root to the file. For example, a path lease on file (4) in Fig. 3 would cover the chain $\{\langle \rangle, \langle 2 \rangle, \langle 2.1 \rangle, \langle 4 \rangle \}$. Path leases are recursive, in that they are issued to other files, specifically to the children of the file whose path is being leased; a path lease on a file can be issued only when the file holds a path lease from its parent. A path lease is accessible to the server that manages the file holding the lease, and if the file is delegated to another server, its path lease migrates with it. The recursive nature of path leases makes them scalable; in particular, the root server need only deal with its immediate child servers, not with every interested party in the system.

When a rename operation causes a server to change the sequence of files along a path, the server must recall any relevant path leases before making the change, which in turn recalls dependent path leases, and so on down the entire subtree. The time to perform a rename thus correlates with the size of the subtree whose root is being renamed. This implies that renames near the root of a large file system may take considerable time to execute. Such renames have considerable semantic impact, so this slowness appears unavoidable.

5 Hotspot mitigation

No one, including us, has ever deployed a file system with no user-visible partitions at our target scale of $\sim 10^5$ clients, so we do not know how such a system would be used in practice. Nonetheless, to plausibly argue that our system can reach such a scale, we believe it necessary to address the problem of workload hotspotting.

At the scales of existing deployments, file-system workloads exhibit little sharing, and this sharing is mainly commutative, meaning that the comparative ordering of two clients' operations is not significant [3, 20]. However, even non-commutative sharing can result in hotspotting if the metadata structure induces false sharing. We avoid false sharing by means of file-field leases and disjunctive leases.

5.1 File-field leases

Rather than a single lease over all metadata of a file, Farsite has a lease over each metadata *field*. The fields are each file attribute, a hash of the file content, the file's deletion disposition [16], a list of which clients have handles open, lists of which clients have the file open for each of Windows' access/locking modes [16], the file identifier of the file's parent, and the child file identifier corresponding to each child name. For the latter fields, since there are infinitely many potential child names, we have a shorthand representation of lease permission over all names other than an explicitly excluded set; we call this the *infinite child* lease.

File-field leases are beneficial when, for example, two clients edit two separate files in the same directory using GNU Emacs [36], which repeatedly creates, deletes, and renames files using a primary name and a backup name. Even though create and rename operations modify the directory's metadata, different filenames are used by each client, so the clients access different metadata fields of the directory.

5.2 Disjunctive leases

For some fields, even a lease per field can admit false sharing. In particular, this applies to the field defining which clients have handles open and to the fields defining which clients have the file open for each of Windows' access/locking modes. Unlike Unix's advisory file locking, file locking in Windows is binding and integral with open and close operations [16]. So, to process an open correctly, a client must determine whether another client has the file open for a particular mode.

For example, if some client *X* has a file open for read access, no other client *Y* can open the file with a mode that excludes others from reading. So, to know whether a "read-exclusive" open operation should result in an error, client *Y* requires read access to the file's "read-mode" field to see if it contains any client identifiers. If, at the same time, client *Z* opens the file for read access, it will require write access to the read-mode field, which conflicts with client *Y*'s read access. This is a case of false sharing, because *Y*'s and *Z*'s opens are semantically commutative with each other.

In Farsite, this false sharing is avoided by applying *disjunctive leases* [12] to each the above fields. For a disjunctive leased field, each client has a Boolean *self* value that it can write and a Boolean *other* value that it can read. The other value for each client x is defined as:

$$other_x = \sum_{y \neq x} self_y$$

where the summation symbol indicates a logical OR.

In the example above, when client *X* opens the file for read access, it sets its self value for the read-mode field to TRUE, which causes client *Y*'s other value to be

TRUE, informing Y that some client has the file open for read access. When client Z opens the file for read access, it sets its self value to TRUE, but this does not change the other value that Y sees.

Each client's self value is protected by a write lease, and its other value is protected by a read lease. The server manages these leases to allow a client to access its values when this does not conflict with the accesses of other clients. For example, if client *X* does not hold a self-write lease, it is not allowed to change its self value; if this value is currently TRUE, then the server can simultaneously grant other-read access to client *Y* and self-write access to client *Z*.

Disjunctive leases are beneficial for, for example, a popular Microsoft Word [18] document. By default, Word attempts to open a file for exclusive access, and if it fails, it then opens the file with shared access. All clients after the first can read and write the appropriate disjunctive mode bits without forcing the server to recall other clients' leases.

6 Design lessons learned

This section enumerates some lessons that we learned while designing Farsite's distributed directory service.

A single field has a single meaning. It might seem natural that if a client has a handle open for write access on a file, the client has the ability to write the file's content. However, this does not follow. It is semantically permissible for two clients to have write-access handles open concurrently, but it is not logistically permissible for them to hold write leases on the file content concurrently. At one point, we attempted to capture these separate meanings in a single field to "simplify" the system, but this actually produced greater complexity and subtlety.

Authority is rigorously defined. In a distributed system wherein servers delegate authority to other servers and lease authority to clients, it is crucial to define which data is authoritative. When we were careless about this, it led to circularities. For example, we once had a write lease that granted authority over all unused names for a file's children; however, whether a name is unused is determined by the machine that has authority over the corresponding metadata field, which is the client if and only if it holds a write lease. In our final design, the lease that grants authority over all-buta-few child names now explicitly identifies the names it excludes, rather than implicitly excluding names that are used.

The lease mechanism is not overloaded with semantics. In one of our designs, if a client tried to open a file that was already open for exclusive access by another client, the server would refuse to issue the client a lease, which the client would interpret as an indication of a mode conflict. Not only does this introduce a special case into the lease logic, but it also

obviates a key benefit of leases: If the client repeatedly tries to open the file, it will repeatedly ask the server for a lease, and the server will repeatedly refuse until the file is closed by the other client. By contrast, mode conflicts are now indicated by issuing the client an other-read disjunctive lease (§ 5.2) on the incompatible mode. With this lease, clients can locally determine whether the open operation should succeed.

Operations are performed on clients and replayed on servers. This arrangement is common for file-content writes in prior distributed file systems, and we found it just as applicable to metadata updates. In an early design, we had considered abandoning this strategy for multi-server rename operations, because we thought it might be simpler for the servers to somehow perform the rename "on behalf of" the client. In testing design alternatives, we found that the regular approach was simpler even for multi-server rename, because it provides a simple operational model: The client obtains leases from all relevant servers, performs the operation locally in a single logical step, and lazily sends the update to the servers. For multi-server operations, the servers coordinate the validation of updates (§ 4.2).

Message recipients need not draw inferences. In one of our design variants, when a client released a lease, it did not indicate to the server whether it was releasing a read lease or a write lease. Because these leases are mutually exclusive, the server could infer which type of lease the client was releasing. However, although this inference was always logically possible, it added some subtlety in a few corner cases, which led to design bugs. The lease-release message now explicitly identifies the lease, because we found that such changes vastly simplify program logic with only a small additional expense in message length.

Leases do not indicate values; they indicate knowledge of values. Before our design was explicit about the fact that leases convey knowledge, we had a function called NameIsUsed that retuned TRUE when the client held a lease over a name field with a non-null value. This led us to carelessly write !NameIsUsed(x) when we really meant NameIsUnused(x). It is easier to spot the error when the functions are named IKnowNameIsUsed and IKnowNameIsUnused. This is even more important when the server validates a client's update with logic such as, "I know that the client knew that name X was unused."

7 Implementation

7.1 Code structure

The directory-service code is split into two main pieces: application logic and an atomic-action substrate. The substrate provides an interface for executing blocks of application code with logical atomicity, thus relegating all concerns about intra-machine race conditions to an

application-generic substrate of about 5K semicolon-lines in C++.

We developed the application logic in the TLA+ system-specification language [22] and then translated it into about 25K semicolon-lines of C++, nearly half of which is data-structure definitions and support routines that were mechanically extracted from the TLA+ spec.

Within the application logic, all aspects of mobility are handled in a bottom layer. Higher-layer client code is written to obtain leases "from files" and to send updates "to files," not from and to servers.

The directory service does not currently employ Byzantine fault tolerance, but the code is structured as a deterministic state machine, so we expect integration with Farsite's existing BFT substrate to be relatively straightforward. As a consequence of the lack of BFT, a machine failure in the current implementation can damage the name space, which is clearly not acceptable for a real deployment.

7.2 Provisional policies

Our emphasis has been on mechanisms that support a seamlessly distributed and dynamically partitioned file system. We have not yet developed optimal policies for managing leases or determining when servers should delegate metadata. For our experimental evaluation, we have developed some provisional policies that appear to work reasonably well.

7.2.1 Lease-management policies

Servers satisfy lease requests in first-come/first-serve order. If a lease request arrives when no conflicting lease is outstanding and no conflicting request is ahead in the queue, the request is satisfied immediately.

For efficiency, a server might choose to promote a set of child-field leases to an infinite child lease (§ 5.1). Our current policy never does so; infinite child leases are issued only for operations that need to know that the file has no children, namely delete and close.

The procedures for managing disjunctive leases (§ 5.2) are fairly involved, as detailed in our tech report [12]. There are two cases that present policy freedom, both of which allow the option of recalling self-write leases from some clients, in the hope that a client's self value is TRUE, thereby coercing all remaining clients' other values to TRUE. Our current policy never exercises this option. When processing an other-read request, we recall all self-write leases, and when processing a self-write request, we recall other-read leases.

7.2.2 Delegation policy

A server considers a file to be *active* if the file's metadata has been accessed within a five-minute interval. The load on a region of file-identifier space is the count of active files in the region. Machines

periodically engage in a pairwise exchange of load information, and a machine that finds its load to be greater than twice the load on another machine will delegate to the less-loaded machine.

To minimize fine-grained repartitioning, which can reduce the efficiency of the file map (§ 4.1.4), our policy uses hysteresis to avoid excessive rebalancing, and it transfers load in a few, heavily-loaded subtrees rather than in many lightly-loaded subtrees.

8 Evaluation

We experimentally evaluate our directory service in a system of 40 machines driven by microbenchmarks, file-system traces, and a synthetic workload. We report on the effectiveness of our techniques as well as the ability of those techniques to seamlessly distribute and dynamically repartition metadata among servers.

8.1 Experimental configuration

We ran our experiments on 40 slightly out-of-date machines reclaimed from people's desktops. Farsite requires no distinction between client and server machines, but for clarity of analysis, we designate half of the machines as clients and half as a pool for servers; this allows us to vary their counts independently. The machines are a mix of single- and dual-processor PIII's with speeds ranging from 733 to 933 MHz and memory from 512 to 1024 MB.

Because Farsite's servers are intended to run on random people's desktop machines, we do not want the server process to consume a significant fraction of a machine's resources. In a real deployment, we might adaptively regulate the server process to keep it from interfering with the user's local processes [10]. For purposes of our experiments, we simulate the limited usability of server machine resources by restricting each server's rate of disk I/O (Farsite's bottleneck resource) to little more than what is needed to support the metadata workload generated by a single client.

8.2 Workloads

We evaluate our system with six workloads: a server snapshot, three microbenchmarks, a file-system trace from a desktop machine, and a synthetic workload that mimics a set of eight user tasks.

8.2.1 Departmental server snapshot

Our simplest workload copies the metadata of a departmental file server into Farsite. The file server contains 2.3M data files and 168K directories.

8.2.2 Conflicting renames

This microbenchmark performs a repeating sequence of four renames issued by two clients, as illustrated in Fig. 1. Client *X* attempts to transform Fig. 1a into Fig. 1b and back again, while client *Y* attempts to transform Fig. 1a into Fig. 1c and back again.

8.2.3 Common-directory editing

This microbenchmark has two clients edit two separate files in the same directory. Each client's operations follow the pattern of GNU Emacs [36], as mentioned in Section 5.1. Each save in the editor results in a creation, a deletion, and a rename, all of which modify child fields in the directory's metadata.

8.2.4 Common-file opening

This microbenchmark has a client create a file using shared-read/exclusive-write access. Then, 12 clients each repeatedly open and close the file, in the manner mentioned in Section 5.2: Each open is first attempted with shared-read/exclusive-write access, and when this fails, the open is repeated with shared-read access.

8.2.5 Desktop file-system trace

To demonstrate a real client workload, we use a one-hour trace of file-system activity. This particular hour within this particular trace was selected because it exhibits the statistically most-typical activity profile of 10am-to-5pm file-system usage among 100 developers' desktop machines we instrumented at Microsoft.

We also drove the system with 15 desktop traces running on 15 clients concurrently. However, multiple traces from independent desktop machines exhibit no sharing, so such an experiment demonstrates no more than a single trace does.

8.2.6 Synthetic workload

To demonstrate a workload with a moderate rate of shared metadata access, we use a synthetic workload driver that models file-system workload with *gremlins*, each of which mimics a user task according to random distributions. Four of the gremlins simulate tasks that exhibit no sharing: entering data into a database; editing text files in GNU Emacs [36]; editing, compiling, and linking a code project; and downloading, caching, and retrieving web pages. Each client runs an instance of all four gremlins, with their control parameters set to make their collective activity roughly match the rate and profile of the desktop file-system trace described above.

To this mix, we add four additional types of gremlins that coordinate among multiple clients: One gremlin simulates shared document editing, with the active reader and/or writer changing every ~3 minutes. Another gremlin simulates an email clique using a maildir [40] file-system-based email system; each client begins a session every ~15 minutes, reads all new email with ~15-second peruse times, and sends ~2 new messages with ~2-minute composition times. A third

gremlin simulates an RCS [39] revision-control system; each client begins a session every ~30 minutes, checks out (co), edits ~5 files, and checks in (ci). Lastly, a custodial gremlin, once per ~30 minutes, renames directories near the root of the file-system tree, above where the other gremlins are working. All indicated times and counts are means of random distributions.

We choose a 15-client/15-server configuration as a basis, and we also vary the count of clients and servers. For each experiment with N clients, we instantiate one custodial gremlin and N of each other gremlin. We run the gremlin workload for 4 hours.

In the basis experiment, the 15 clients performed 390 operations/second in aggregate. This resulted in an aggregate rate of 3.6 lease requests, 8.3 lease grants, and 0.8 batched updates per second.

8.3 Results

This section reports on the effectiveness of our main techniques: tree-structured file identifiers, multi-server operations, path leases, file-field leases, and disjunctive leases. It also reports on the results of those techniques, namely the load balancing, proportional scaling, and semantic correctness of our directory service.

8.3.1 Tree-structured file identifiers

To validate our hypothesis that tree-structured file identifiers can compactly represent real directory-size and directory-depth distributions, we drive the system with a server snapshot (§ 8.2.1). New file identifiers are created according to Farsite's rule for keeping files that are close in the name space also close in the identifier space (§ 4.1.5). As a control, we use simulation to evaluate the use of conventional fixed-size identifiers to satisfy this same condition. Specifically, our control uses non-oracle greedy assignment, which successively halves sub-regions of the flat identifier space for each new subdirectory in a directory, and it assigns file identifiers to files in a directory linearly within the directory's sub-region of identifier space. When the optimal sub-region is filled up, new identifiers spill into ancestor regions.

In the experimental run, the mean identifier length is 52 bits, the maximum is 107 bits, and the 99th percentile is 80 bits. Because file identifiers grow logarithmically with file-system size, a much larger file system would produce only slightly larger identifiers.

We performed three control runs. With 32-bit file identifiers, only 16% of the server's files were assigned identifiers within their designated sub-region, and only the first 62% were assigned identifiers at all, because eventually there were no free identifiers on the path from the parent identifier all the way to the root. When the file-identifier length is increased to 48 bits, which is the same size as the immediate representation of our

tree-structured file identifiers, all files obtained identifiers; however, 46% of them spilled outside their designated sub-region. Even with 107-bit identifiers, which is the maximum size tree-structured identifier needed for this workload, 5% of files spilled outside their designated sub-region. Moreover, any fixed-size identifier scheme has a spillage cliff that will eventually be reached as the file system grows.

8.3.2 Multi-server operations and path leases

To characterize the multi-server rename protocol and its use of path leases, we drive the system with a worst-case workload of conflicting renames (§ 8.2.2), wherein all six relevant files are on different servers.

Over 400 rename operations, the run produced 2404 messages relating to requesting, recalling, issuing, and releasing path leases. It also produced 2400 interserver messages related to the two-phase locking in our multi-server move protocol. On average, there were 12 server-to-server messages per rename operation in this extreme worst case.

8.3.3 File-field leases

To validate our hypotheses that file-field leases prevent an instance of false sharing, we drive the system with a workload of common-directory editing (§ 8.2.3). As a control, we replace file-field leases with a single lease per file.

After 100 editor saves, the control run issued 202 file leases, whereas the experimental run issued 6 child-field leases. In the control run, the count of leases is proportional to editor saves, but in the experimental run, the count of leases is constant.

The synthetic workload (§ 8.2.6) also demonstrates the benefit of file-filed leases. Of the 110k leases the servers issued, 59% did not require recalling other leases but would have required recalls if the lease conflicts had been computed per file. This actually understates the benefit, because a client whose lease has been recalled is likely to want it back, leading to an arbitrarily greater degree of lease ping-ponging.

8.3.4 Disjunctive leases

To validate our hypotheses that disjunctive leases prevent an instance of false sharing, we drive the system with a workload of common-file opening (§ 8.2.4). As a control, we replace each disjunctive lease with a single-writer/multiple-reader lease.

After 100 iterations per client, the control run issued 1237 mode-field leases and recalled 1203 of them, whereas the experimental run issued 13 mode-field leases and recalled one. In the control run, the lease count is proportional to client open attempts, but in the experimental run, the lease count is merely proportional to the count of clients.

8.3.5 Load balancing

To validate our hypothesis that dynamic partitioning effectively balances metadata load, we look at the loads on the 15 server machines in a basis run of the synthetic workload (§ 8.2.6). Fig. 4 shows the loads over the first 90 minutes of the 4-hour run. Initially, the entire load is on the root server, but this is quickly delegated away to other servers. If the load were perfectly balanced, each server would handle 1/15th of the load, but the actual load on machines varies from 3% to 14%, indicating that there is room for improvement in our delegation policy.

As a control, we modify our delegation policy to delegate each file at most once. This approximates an alternative system design that attempts to balance load by statically assigning each file to a server when the file is created. We could not implement this alternative directly, because Farsite assumes that a file is created on the same server as its parent. However, our control is a conservative approximation, because it can look at a file's activity for an arbitrary period after creation before deciding on an appropriate server for the file. Fig. 5 shows the loads for the control, which initially does a passable job of balancing the loads, but they become progressively less balanced as time goes on.

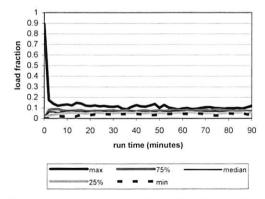


Fig. 4: Server Load vs. Time, Full Delegation

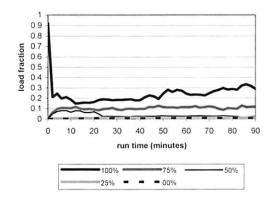


Fig. 5: Server Load vs. Time, Limited Delegation

8.3.6 Proportional scaling

A key aspect of Farsite's deployment scenario is that every client can also be a server. To demonstrate that Farsite is able to use a proportional server pool effectively, we ran the synthetic workload (§ 8.2.6) in a suite of experiments with N clients and N server machines, varying N from 1 to 20. Fig. 6 illustrates the mean and median client throughput relative to the 15client/15-server basis described above. The graph shows that per-client throughput holds approximately constant as the system size varies; there is some random variation, but it is not correlated to system scale. This experiment shows that, at least at these modest scales, our mechanisms are effective at preventing the root server (or any other resource) from becoming a bottleneck that causes a decrease in throughput as scale increases.

For contrast, we ran a suite of experiments with *N* server machines but a fixed load of 15 clients, as shown in Fig. 7. As long as the count of servers is not much below the count of clients, the per-client throughput is not noticeably throttled by available server resources. Because we throttle the servers' I/O rates, smaller server pools cause a drop in client throughput. This illustrates the need for distributing the directory service in Farsite.

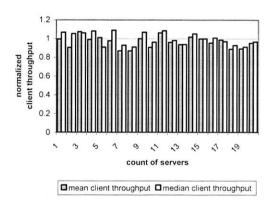


Fig. 6: Throughput, Equal Clients and Servers

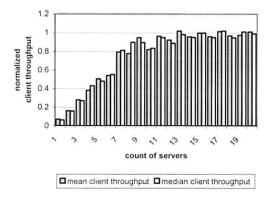


Fig. 7: Throughput, 15 Clients

8.3.7 Semantic correctness

Experiments with all of the workloads demonstrate that our mechanisms successfully preserve file-system semantics, even when the metadata is dynamically partitioned among servers. In the file-system trace workload (§ 8.2.5), the results of the submitted operations match the results when the trace is run on a local file system. The microbenchmarks (§§ 8.2.2–4) and the gremlins (§ 8.2.6) know what the results of their operations should be, and they verify that the system functions correctly.

In the gremlin basis experiment, 8 operations spanned multiple servers. Although this is a small fraction of the 2.1M operations performed, even a single semantic failure is enough to annoy a real user. If Farsite's metadata partitioning were user-visible, these 8 operations could not have been performed.

9 Future work

We see several areas for future work, including further analysis, additional mechanism, improved policy, and completing integration with the rest of Farsite.

9.1 Further analysis

Our synthetic workload (§ 8.2.6) exhibits a moderate degree of shared metadata access, as indicated by measurements of distributed-file-system usage [3]. It would be valuable to study the sensitivity of the system to variations in the degree of sharing.

We would like to study Farsite's behavior at scales far larger than we could test in our lab. Simulation may help to understand the behavior of our distributed algorithms at the target scale of 10⁵ machines.

9.2 Mitigating super-hotspots

If a very large number of clients request concurrent, non-conflicting access to a particular metadata field, the managing server may become overwhelmed by the load of issuing and recalling leases. We envision diffusing this load with a *blanket lease*: a signed certificate that declares the field to have a certain value guaranteed for a certain time, thus effectively granting every client read access to the field. Whenever the count of lease requests exceeds a threshold, a blanket lease is sent to all connected clients via a distribution tree.

Because such leases cannot be recalled, if a client requests a lease that conflicts with a blanket lease, the server must wait for the blanket lease to expire before it can issue the conflicting lease. To prevent the clients that had been using the blanket lease from hammering the server with new lease requests, the server distributes a *stifle certificate*, which tells the clients not to make any requests for the metadata until an indicated time,

thus giving the server a chance to satisfy the conflicting request and then issue a new blanket lease.

9.3 Improved delegation policy

Our current delegation policy (§ 7.2.2) attempts to balance the distribution of load, under some basic constraints to minimize fragmentation. There is much opportunity both to improve efficiency and to reduce fine-grained repartitioning.

The delegation policy could also be extended to account for variations in machine availability, similar to Farsite's policy for file-content placement [11].

9.4 Complete system integration

The directory service is not yet fully integrated with the rest of the Farsite system. In particular, there is no bridge to the components that manage file content. The main hurdle is modifying Farsite's kernel-level file-system driver to understand the leases used by the distributed directory service, which differ considerably from those used by Farsite's earlier centralized directory service.

The directory service code is not currently built on top of Farsite's BFT substrate. However, we structured the code with BFT in mind, so this integration should not be a significant challenge.

10 Related work

For comparison with our system, a convenient way to categorize distributed file systems is by the method they use to partition metadata.

10.1 Dynamically partitioning the name tree

Weil et al. [42] propose a file-system metadata service that dynamically partitions metadata by subtrees of the name space. They evaluate this proposal via simulation rather than by implementation and experimentation. Despite their use of simulation, they only study configurations up to 50 machines, which is not much larger than our evaluation. They conclude that subtree partitioning is more efficient than either hash-based metadata partitioning or a hybrid of the two.

Although their proposed partitioning scheme bears similarity to ours, a key difference is that our dynamic partitioning is based on tree-structured file identifiers rather than the name-space tree, which we argue (§ 4) poses significant challenges for rename operations. It is not clear that Weil et al.'s simulations assess the cost of performing renames in their proposed system.

10.2 Statically partitioning the name tree

Several distributed file systems partition their metadata statically by path name. Examples include NFS and the

Automounter from Sun, CIFS and Dfs from Microsoft, and Sprite.

NFS [31] and CIFS [17] both afford remote access to server-based file systems, but neither one provides a global name space. A collection of NFS "mountpoints" can be made to appear as a global name space when mounted uniformly with the client-side Automounter [5]. A collection if CIFS "shares" can be assembled into a global name space by the Dfs [25] client/server redirection system. NFS and CIFS both require all client operations to be sent to the server for processing. The Automounter and Dfs both operate on path names.

Sprite [27] partitions the file-system name space into "domains." Clients find the managing server for a domain initially by broadcast and thereafter by use of a prefix table [43], which maps path names to servers according to longest-matching-prefix. Farsite's file map (§ 4.1.4) is similar to Sprite's prefix table, except that it operates on file identifiers rather than path names.

Mountpoints, shares, and domains are all uservisible. It is not possible to perform rename operations across them, unlike Farsite's transparent partitions. Furthermore, none of these systems provides support for automatic load balancing.

10.3 Partitioning groups of file identifiers

AFS [19] partitions files using a hybrid scheme based partly on file name and partly on file identifier. As in Sprite, the name space is statically partitioned by path name into partial subtrees. Each subtree, known as a "volume," is dynamically assigned to a server. Files are partitioned among volumes according to file identifier, which embeds a volume identifier. Like Sprite's shares, AFS volumes are user-visible with regard to rename.

xFS [2] partitions metadata by file identifier, which it calls the "file index number." A globally replicated manager map uses a portion of a file's index number to determine which machine manages the file's metadata. All files whose index numbers correspond to the same manager-map entry are managed by the same machine, so they correspond closely to an AFS volume, although xFS presents no name for this abstraction. Whereas AFS volumes provide name-space locality, xFS files are partitioned according to which client originally created the file. Relocating groups of files, which requires consistently modifying the global manager map, is not implemented in the xFS prototype, xFS is intended to have no user-visible partitions, but the paper includes no discussion of the rename operation or of any name-space consistency mechanisms.

In both AFS and xFS, fine-grained repartitioning of files requires changing files' identifiers, which entails significant administrator intervention.

10.4 Partitioning by hashing file names

Another way to partition metadata is to use a distributed hash table. CFS [8] and PAST [29] are scalable, distributed storage systems based, respectively, on the Chord [37] and PASTRY [30] distributed hash tables. CFS and PAST provide only flat name spaces, unlike the hierarchical name space provided by Farsite.

10.5 Partitioning at the block level

A markedly different approach is commonly employed by cluster file systems, which construct a distributed, block-level storage substrate for both file content and metadata. The metadata is partitioned at the level of the block-storage substrate. At the file-system level, all metadata is accessible to all servers in the cluster, which use a distributed lock manager to coordinate their metadata accesses.

A typical example is the Global File System [35], which builds a "network storage pool" from a diverse collection of network-attached storage devices. High-speed connectivity between all servers and all storage devices is provided by a storage area network (SAN). Servers do not communicate directly with each other; rather, lock management of shared storage devices is provided by storage device controllers.

Instead of running over a SAN, the Frangipani [38] file system is built on the Petal [24] distributed virtual disk, which pools the storage resources of multiple server machines into logically centralized disk. High-speed connectivity is provided by a switched ATM network. Frangipani servers use a message-based lock-management scheme, and they maintain consistent configuration information using Paxos [21].

Because these systems do not partition metadata, their performance can suffer due to contention. IBM's GPFS [33] introduces several techniques to decrease the likelihood of contended metadata access. First, shared-write locks allow concurrent updating of nonname-space-critical metadata. Specifically, updates to file size and modification time are committed lazily. Second, the allocation map is divided into large regions. A single allocation manager server keeps the allocation map loosely up-to-date and directs different servers to different regions of the map. Deletions are shipped to the servers managing the relevant regions of the map. Third, distributed lock management employs a token protocol. Although this is supervised by a central token manager, several optimizations reduce the message load, including batching, prefetching, and hysteresis.

These systems all require tight coupling via SAN or switched network, unlike Farsite, which runs over a LAN. Because their metadata is partitioned at the level of opaque blocks, lock contention can limit the scalability of some metadata operations [33 (Fig. 4)].

10.6 Not partitioning

The Google File System [14] does not partition metadata at all. All metadata updates are centralized on a single server. The metadata workload is kept low by providing a limited operational interface for appendonly or append-mostly applications that can tolerate duplicated writes.

11 Summary and conclusions

We have developed a distributed directory service for Farsite [1], a logically centralized file system physically distributed among a wired network of desktop machines on the campus of a large corporation or university. Farsite provides the benefits of a central file server without the additional hardware cost, susceptibility to geographically localized faults, and frequent reliance on system administrators entailed by a central server.

Prior to this work, Farsite's metadata service was centralized on a single BFT group of machines, which could not scale to the normal file-system metadata loads [41] of the ~10⁵ desktop computers [4] in our target environment. In designing a distributed replacement for this service, a key goal was to automate the balancing of workload among servers, so as to minimize the need for human system administration.

A significant concern was that as file metadata is dynamically relocated among servers to balance load, users might observe confusingly varying semantics over time. To avoid this situation, we designed our directory service so that the partitioning of files among servers is not user-visible. Specifically, Farsite supports a fully functional rename operation, which allows renames to be performed anywhere in the name space, unlike in previous distributed file systems [19, 25, 27]. Since a rename operation may thus span multiple servers, the servers employ two-phase locking to coordinate their processing of a rename.

The name space is strongly consistent. Although necessary only for rename to prevent the accidental disconnection of parts of the directory tree, strong consistency is maintained for all path-based operations. To avoid the scalability limitation of having servers issue leases on their files' children to all interested parties, we employ recursive path leases, which are successively issued from each file to its child files, and which inform a file of its current path name.

We partition files among servers according to file identifier, because the alternative of partitioning by file name is complicated by the mutability of names under the rename operation. Our file identifiers have a tree structure that stays approximately aligned with the tree structure of the name space, so files can be efficiently partitioned with arbitrary granularity while making few cuts in the name space.

To mitigate the hotspots that might arise in a deployed system, our service has a separate lease over each metadata field of a file, rather than a single lease over all metadata of a file. For certain fields relating to Windows' deletion semantics and access/locking modes [16], we break down fields even further by means of disjunctive leases, wherein each client supplies an independent Boolean value for its part of the field, and each client observes the logical OR of other clients' values. With the exception of disjunctive leases, all of our techniques are directly applicable to non-Windows file systems.

Although our emphasis was on mechanisms, we also developed some provisional policies to drive those mechanisms. For managing leases, we used the simplest approaches we could: issuing leases on a first-come/first-serve basis, promoting leases only when necessary, and pessimistically recalling all potentially conflicting disjunctive leases. To relocate file metadata, machines attempt to balance recent file activity, using a pairwise exchange with hysteresis.

We experimentally evaluated our service in a system of 40 machines, using a server snapshot, microbenchmarks, traces, and a synthetic workload. The snapshot shows that tree-structured file identifiers can compactly represent a real file-server directory structure. Microbenchmarks show that file-field leases and disjunctive leases mitigate hotspots by preventing instances of false sharing. The trace and synthetic workloads show that dynamic partitioning effectively balances metadata load. They also show that the system workload capacity increases in proportion to the system size, which is a crucial property for achieving our target scale of ~10⁵ machines.

Acknowledgements

The authors deeply appreciate the long-running support and encouragement this effort has received from the other members of the Farsite team: Atul Adya, Bill Bolosky, Miguel Castro, Ronnie Chaiken, Jay Lorch, and Marvin Theimer.

The effort of distilling our work into a presentable form has been assisted by many people, beginning with Butler Lampson, who helped us hone the paper's focus. Early versions of the paper received valuable comments from Rich Draves, John Dunagan, Jeremy Elson, Leslie Lamport, Marc Shapiro, Marvin Theimer, Helen Wang, Lidong Zhou, and Brian Zill. The final version has benefited greatly from the many detailed comments and suggestions made by the anonymous OSDI reviewers and especially from the meticulous attention of our shepherd, Garth Gibson.

We also extend our thanks to Robert Eberl of MSR Tech Support for his help in collecting the departmental server snapshot described in Section 8.2.1.

References

- [1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, R. P. Wattenhofer. "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," 5th OSDI, Dec 2002.
- [2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, R. Wang. "Serverless Network File Systems," 15th SOSP, 1995.
- [3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, J. K. Ousterhout. "Measurements of a Distributed File System," 13th SOSP, 1991.
- [4] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs," SIGMETRICS 2000, Jun 2000.
- [5] B. Callaghan, T. Lyon. "The Automounter," Winter USENIX Conf. 30 (3), 1989.
- [6] A. Campbell. Managing AFS: The Andrew File System, Prentice Hall, 1998.
- [7] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," 3rd OSDI, Feb 1999.
- [8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, "Wide-Area Cooperative Storage with CFS," 18th SOSP, Oct 2001.
- [9] J. R. Douceur, W. J. Bolosky. "A Large-Scale Study of File-System Contents," SIGMETRICS '99, May 1999.
- [10] J. R. Douceur, W. J. Bolosky. "Progress-Based Regulation of Low-Importance Processes," 17th SOSP, Dec 1999.
- [11] J. R. Douceur, R. P. Wattenhofer, "Optimizing File Availability in a Secure Serverless Distributed File System," 20th SRDS, Oct 2001.
- [12] J. R. Douceur, J. Howell. "Black Box Leases," Microsoft Research Tech Report MSR-TR-2005-120, 2005.
- [13] P. Elias. "Universal Codeword Sets and Representations of the Integers," *IEEE Trans. Info. Theory* 21(2), 1975.
- [14] S. Ghemawat, H. Gobioff, S-T. Leung. "The Google File System," 19th SOSP, 2003.
- [15] C. G. Gray, D. R. Cheriton. "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency," 12th SOSP, 1989.
- [16] J. M. Hart. Win32 System Programming: A Windows(R) 2000 Application Developer's Guide, Second Edition, Addison-Wesley, 2000.
- [17] C. R. Hertel. Implementing CIFS: The Common Internet File System, Prentice Hall, 2003.
- [18] B. Heslop, D. Angell, P. Kent. Word 2003 Bible, Wiley, 2003.
- [19] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, M. West. "Scale and Performance in a Distributed File System," *TOCS* 6(1), Feb 1988.
- [20] J. Kistler, M. Satyanarayanan. "Disconnected operation in the Coda File System," *TOCS* 10(1), Feb 1992.
- [21] L. Lamport. "The Part-Time Parliament," TOCS 16(2), 1998
- [22] L. Lamport. Specifying Systems. Addison-Wesley, 2003.

- [23] B. W. Lampson, V. Srinivasan, G. Varghese. "IP Lookups using Multiway and Multicolumn Search," *IEEE/ACM Trans. Networking* 7(3), 1999.
- [24] E. Lee, C. Thekkath. "Petal: Distributed Virtual Disks," 7th ASPLOS, 1996.
- [25] T. Northrup. NT Network Plumbing: Routers, Proxies, and Web Services, IDG Books, 1998.
- [26] D. Oppenheimer, A. Ganapathi, D. A. Patterson. "Why Do Internet Services Fail, and What Can Be Done About It?" 4th USITS, Mar 2003
- [27] J. K. Ousterhout, A, R. Cherenson, F. Douglis, M. N. Nelson, B. B. Welch. "The Sprite Network Operating System," *IEEE Computer Group Magazine* 21 (2), 1988.
- [28] T. Rizzo. Programming Microsoft Outlook and Microsoft Exchange 2003, Third Edition, Microsoft Press, 2003.
- [29] A. Rowstron and P. Druschel. "Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility", 18th SOSP, Oct 2001.
- [30] A. Rowstron, P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems", *Middleware 2001*, Nov 2001.
- [31] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon. "Design and Implementation of the Sun Network File System," USENIX 1985 Summer Conference, 1985.
- [32] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, M. J. West. "The ITC Distributed File System: Principles and Design," 10th SOSP, Dec 1985.
- [33] F. Schmuck, R. Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters," *1st FAST*, 2002.
- [34] B. Sidebotham. "Volumes: The Andrew File System Data Structuring Primitive," EUUG Conference Proceedings, Aug 1986.
- [35] S. R. Soltis, G. M. Erickson, K. W. Preslan, M. T. O'Keefe, T. M. Ruwart. "The Global File System: A File System for Shared Disk Storage," 1997 http://www.diku.dk/undervisning/2003e/314/papers/ soltis97global.pdf
- [36] R. M. Stallman. *GNU Emacs Manual, For Version 21, 15th Edition,* Free Software Foundation, 2002.
- [37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," SIGCOMM 2001, Aug 2001.
- [38] C. Thekkath, T. Mann, E. Lee. "Frangipani: A Scalable Distributed File System," 16th SOSP, Dec 1997.
- [39] W. Tichy. "RCS: A System for Version Control," Software–Practice & Experience 15(7), 1985.
- [40] Sam Varshavchik. "Benchmarking mbox versus maildir," 2003 http://www.courier-mta.org/mbox-vs-maildir/
- [41] W. Vogels. "File System Usage in Windows NT 4.0," 17th SOSP, Dec 1999.
- [42] S. A. Weil, K. T. Pollack, S. A. Brandt, E. L. Miller. "Dynamic Metadata Management for Petabyte-Scale File Systems," *Supercomputing* 2004, 2004.
- [43] B. Welch, J. Ousterhout. "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System," 6th ICDCS, 1986.

The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, Google Inc.

Abstract

We describe our experiences with the Chubby lock service, which is intended to provide coarse-grained locking as well as reliable (though low-volume) storage for a loosely-coupled distributed system. Chubby provides an interface much like a distributed file system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance. Many instances of the service have been used for over a year, with several of them each handling a few tens of thousands of clients concurrently. The paper describes the initial design and expected use, compares it with actual use, and explains how the design had to be modified to accommodate the differences.

1 Introduction

This paper describes a *lock service* called Chubby. It is intended for use within a loosely-coupled distributed system consisting of moderately large numbers of small machines connected by a high-speed network. For example, a Chubby instance (also known as a Chubby *cell*) might serve ten thousand 4-processor machines connected by 1Gbit/s Ethernet. Most Chubby cells are confined to a single data centre or machine room, though we do run at least one Chubby cell whose replicas are separated by thousands of kilometres.

The purpose of the lock service is to allow its clients to synchronize their activities and to agree on basic information about their environment. The primary goals included reliability, availability to a moderately large set of clients, and easy-to-understand semantics; throughput and storage capacity were considered secondary. Chubby's client interface is similar to that of a simple file system that performs *whole-file* reads and writes, augmented with advisory locks and with notification of various events such as file modification.

We expected Chubby to help developers deal with coarse-grained synchronization within their systems, and in particular to deal with the problem of electing a leader from among a set of otherwise equivalent servers. For example, the Google File System [7] uses a Chubby lock to appoint a GFS master server, and Bigtable [3] uses Chubby in several ways: to elect a master, to allow the master to discover the servers it controls, and to permit clients to find the master. In addition, both GFS and Bigtable use Chubby as a well-known and available location to store a small amount of meta-data; in effect they use Chubby as the root of their distributed data structures. Some services use locks to partition work (at a coarse grain) between several servers.

Before Chubby was deployed, most distributed systems at Google used *ad hoc* methods for primary election (when work could be duplicated without harm), or required operator intervention (when correctness was essential). In the former case, Chubby allowed a small saving in computing effort. In the latter case, it achieved a significant improvement in availability in systems that no longer required human intervention on failure.

Readers familiar with distributed computing will recognize the election of a primary among peers as an instance of the distributed consensus problem, and realize we require a solution using asynchronous communication; this term describes the behaviour of the vast majority of real networks, such as Ethernet or the Internet, which allow packets to be lost, delayed, and reordered. (Practitioners should normally beware of protocols based on models that make stronger assumptions on the environment.) Asynchronous consensus is solved by the Paxos protocol [12, 13]. The same protocol was used by Oki and Liskov (see their paper on viewstamped replication [19, §4]), an equivalence noted by others [14, §6]. Indeed, all working protocols for asynchronous consensus we have so far encountered have Paxos at their core. Paxos maintains safety without timing assumptions, but clocks must be introduced to ensure liveness; this overcomes the impossibility result of Fischer et al. [5, §1].

Building Chubby was an engineering effort required to fill the needs mentioned above; it was not research. We claim no new algorithms or techniques. The purpose of this paper is to describe what we did and why, rather than to advocate it. In the sections that follow, we describe Chubby's design and implementation, and how it

has changed in the light of experience. We describe unexpected ways in which Chubby has been used, and features that proved to be mistakes. We omit details that are covered elsewhere in the literature, such as the details of a consensus protocol or an RPC system.

2 Design

2.1 Rationale

One might argue that we should have built a library embodying Paxos, rather than a library that accesses a centralized lock service, even a highly reliable one. A client Paxos library would depend on *no* other servers (besides the name service), and would provide a standard framework for programmers, assuming their services can be implemented as state machines. Indeed, we provide such a client library that is independent of Chubby.

Nevertheless, a lock service has some advantages over a client library. First, our developers sometimes do not plan for high availability in the way one would wish. Often their systems start as prototypes with little load and loose availability guarantees; invariably the code has not been specially structured for use with a consensus protocol. As the service matures and gains clients, availability becomes more important; replication and primary election are then added to an existing design. While this could be done with a library that provides distributed consensus, a lock server makes it easier to maintain existing program structure and communication patterns. For example, to elect a master which then writes to an existing file server requires adding just two statements and one RPC parameter to an existing system: One would acquire a lock to become master, pass an additional integer (the lock acquisition count) with the write RPC, and add an if-statement to the file server to reject the write if the acquisition count is lower than the current value (to guard against delayed packets). We have found this technique easier than making existing servers participate in a consensus protocol, and especially so if compatibility must be maintained during a transition period.

Second, many of our services that elect a primary or that partition data between their components need a mechanism for advertising the results. This suggests that we should allow clients to store and fetch small quantities of data—that is, to read and write small files. This could be done with a name service, but our experience has been that the lock service itself is well-suited for this task, both because this reduces the number of servers on which a client depends, and because the consistency features of the protocol are shared. Chubby's success as a name server owes much to its use of consistent client caching, rather than time-based caching. In particular, we found that developers greatly appreciated not having

to choose a cache timeout such as the DNS time-to-live value, which if chosen poorly can lead to high DNS load, or long client fail-over times.

Third, a lock-based interface is more familiar to our programmers. Both the replicated state machine of Paxos and the critical sections associated with exclusive locks can provide the programmer with the illusion of sequential programming. However, many programmers have come across locks before, and think they know to use them. Ironically, such programmers are usually wrong, especially when they use locks in a distributed system; few consider the effects of independent machine failures on locks in a system with asynchronous communications. Nevertheless, the apparent familiarity of locks overcomes a hurdle in persuading programmers to use a reliable mechanism for distributed decision making.

Last, distributed-consensus algorithms use quorums to make decisions, so they use several replicas to achieve high availability. For example, Chubby itself usually has five replicas in each cell, of which three must be running for the cell to be up. In contrast, if a client system uses a lock service, even a single client can obtain a lock and make progress safely. Thus, a lock service reduces the number of servers needed for a reliable client system to make progress. In a loose sense, one can view the lock service as a way of providing a generic electorate that allows a client system to make decisions correctly when less than a majority of its own members are up. One might imagine solving this last problem in a different way: by providing a "consensus service", using a number of servers to provide the "acceptors" in the Paxos protocol. Like a lock service, a consensus service would allow clients to make progress safely even with only one active client process; a similar technique has been used to reduce the number of state machines needed for Byzantine fault tolerance [24]. However, assuming a consensus service is not used exclusively to provide locks (which reduces it to a lock service), this approach solves none of the other problems described above.

These arguments suggest two key design decisions:

- We chose a lock service, as opposed to a library or service for consensus, and
- we chose to serve small-files to permit elected primaries to advertise themselves and their parameters, rather than build and maintain a second service.

Some decisions follow from our expected use and from our environment:

- A service advertising its primary via a Chubby file may have thousands of clients. Therefore, we must allow thousands of clients to observe this file, preferably without needing many servers.
- Clients and replicas of a replicated service may wish to know when the service's primary changes. This

- suggests that an event notification mechanism would be useful to avoid polling.
- Even if clients need not poll files periodically, many will; this is a consequence of supporting many developers. Thus, caching of files is desirable.
- Our developers are confused by non-intuitive caching semantics, so we prefer consistent caching.
- To avoid both financial loss and jail time, we provide security mechanisms, including access control.

A choice that may surprise some readers is that we do not expect lock use to be *fine-grained*, in which they might be held only for a short duration (seconds or less); instead, we expect *coarse-grained* use. For example, an application might use a lock to elect a primary, which would then handle all access to that data for a considerable time, perhaps hours or days. These two styles of use suggest different requirements from a lock server.

Coarse-grained locks impose far less load on the lock server. In particular, the lock-acquisition rate is usually only weakly related to the transaction rate of the client applications. Coarse-grained locks are acquired only rarely, so temporary lock server unavailability delays clients less. On the other hand, the transfer of a lock from client to client may require costly recovery procedures, so one would not wish a fail-over of a lock server to cause locks to be lost. Thus, it is good for coarse-grained locks to survive lock server failures, there is little concern about the overhead of doing so, and such locks allow many clients to be adequately served by a modest number of lock servers with somewhat lower availability.

Fine-grained locks lead to different conclusions. Even brief unavailability of the lock server may cause many clients to stall. Performance and the ability to add new servers at will are of great concern because the transaction rate at the lock service grows with the combined transaction rate of clients. It can be advantageous to reduce the overhead of locking by not maintaining locks across lock server failure, and the time penalty for dropping locks every so often is not severe because locks are held for short periods. (Clients must be prepared to lose locks during network partitions, so the loss of locks on lock server fail-over introduces no new recovery paths.)

Chubby is intended to provide only coarse-grained locking. Fortunately, it is straightforward for clients to implement their own fine-grained locks tailored to their application. An application might partition its locks into groups and use Chubby's coarse-grained locks to allocate these lock groups to application-specific lock servers. Little state is needed to maintain these fine-grain locks; the servers need only keep a non-volatile, monotonically-increasing acquisition counter that is rarely updated. Clients can learn of lost locks at unlock time, and if a simple fixed-length lease is used, the protocol can be simple and efficient. The most important benefits of this

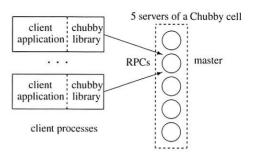


Figure 1: System structure

scheme are that our client developers become responsible for the provisioning of the servers needed to support their load, yet are relieved of the complexity of implementing consensus themselves.

2.2 System structure

Chubby has two main components that communicate via RPC: a server, and a library that client applications link against; see Figure 1. All communication between Chubby clients and the servers is mediated by the client library. An optional third component, a proxy server, is discussed in Section 3.1.

A Chubby cell consists of a small set of servers (typically five) known as *replicas*, placed so as to reduce the likelihood of correlated failure (for example, in different racks). The replicas use a distributed consensus protocol to elect a *master*; the master must obtain votes from a majority of the replicas, plus promises that those replicas will not elect a different master for an interval of a few seconds known as the *master lease*. The master lease is periodically renewed by the replicas provided the master continues to win a majority of the vote.

The replicas maintain copies of a simple database, but only the master initiates reads and writes of this database. All other replicas simply copy updates from the master, sent using the consensus protocol.

Clients find the master by sending master location requests to the replicas listed in the DNS. Non-master replicas respond to such requests by returning the identity of the master. Once a client has located the master, the client directs all requests to it either until it ceases to respond, or until it indicates that it is no longer the master. Write requests are propagated via the consensus protocol to all replicas; such requests are acknowledged when the write has reached a majority of the replicas in the cell. Read requests are satisfied by the master alone; this is safe provided the master lease has not expired, as no other master can possibly exist. If a master fails, the other replicas run the election protocol when their master leases expire; a new master will typically be elected in a few seconds. For example, two recent elections took 6s and 4s, but we see values as high as 30s (§4.1).

If a replica fails and does not recover for a few hours, a simple replacement system selects a fresh machine from a free pool and starts the lock server binary on it. It then updates the DNS tables, replacing the IP address of the failed replica with that of the new one. The current master polls the DNS periodically and eventually notices the change. It then updates the list of the cell's members in the cell's database; this list is kept consistent across all the members via the normal replication protocol. In the meantime, the new replica obtains a recent copy of the database from a combination of backups stored on file servers and updates from active replicas. Once the new replica has processed a request that the current master is waiting to commit, the replica is permitted to vote in the elections for new master.

2.3 Files, directories, and handles

Chubby exports a file system interface similar to, but simpler than that of UNIX [22]. It consists of a strict tree of files and directories in the usual way, with name components separated by slashes. A typical name is:

/ls/foo/wombat/pouch

The 1s prefix is common to all Chubby names, and stands for *lock service*. The second component (foo) is the name of a Chubby cell; it is resolved to one or more Chubby servers via DNS lookup. A special cell name local indicates that the client's local Chubby cell should be used; this is usually one in the same building and thus the one most likely to be accessible. The remainder of the name, /wombat/pouch, is interpreted within the named Chubby cell. Again following UNIX, each directory contains a list of child files and directories, while each file contains a sequence of uninterpreted bytes.

Because Chubby's naming structure resembles a file system, we were able to make it available to applications both with its own specialized API, and via interfaces used by our other file systems, such as the Google File System. This significantly reduced the effort needed to write basic browsing and name space manipulation tools, and reduced the need to educate casual Chubby users.

The design differs from UNIX in a ways that ease distribution. To allow the files in different directories to be served from different Chubby masters, we do not expose operations that can move files from one directory to another, we do not maintain directory modified times, and we avoid path-dependent permission semantics (that is, access to a file is controlled by the permissions on the file itself rather than on directories on the path leading to the file). To make it easier to cache file meta-data, the system does not reveal last-access times.

The name space contains only files and directories, collectively called *nodes*. Every such node has only one name within its cell; there are no symbolic or hard links.

Nodes may be either permanent or ephemeral. Any node may be deleted explicitly, but ephemeral nodes are also deleted if no client has them open (and, for directories, they are empty). Ephemeral files are used as temporary files, and as indicators to others that a client is alive. Any node can act as an advisory reader/writer lock; these locks are described in more detail in Section 2.4.

Each node has various meta-data, including three names of access control lists (ACLs) used to control reading, writing and changing the ACL names for the node. Unless overridden, a node inherits the ACL names of its parent directory on creation. ACLs are themselves files located in an ACL directory, which is a well-known part of the cell's local name space. These ACL files consist of simple lists of names of principals; readers may be reminded of Plan 9's groups [21]. Thus, if file F's write ACL name is £00, and the ACL directory contains a file £00 that contains an entry bar, then user bar is permitted to write F. Users are authenticated by a mechanism built into the RPC system. Because Chubby's ACLs are simply files, they are automatically available to other services that wish to use similar access control mechanisms.

The per-node meta-data includes four monotonicallyincreasing 64-bit numbers that allow clients to detect changes easily:

- an instance number; greater than the instance number of any previous node with the same name.
- a content generation number (files only); this increases when the file's contents are written.
- a lock generation number; this increases when the node's lock transitions from *free* to *held*.
- an ACL generation number; this increases when the node's ACL names are written.

Chubby also exposes a 64-bit file-content checksum so clients may tell whether files differ.

Clients open nodes to obtain *handles* that are analogous to UNIX file descriptors. Handles include:

- check digits that prevent clients from creating or guessing handles, so full access control checks need be performed only when handles are created (compare with UNIX, which checks its permissions bits at open time, but not at each read/write because file descriptors cannot be forged).
- a sequence number that allows a master to tell whether a handle was generated by it or by a previous master.
- mode information provided at open time to allow the master to recreate its state if an old handle is presented to a newly restarted master.

2.4 Locks and sequencers

Each Chubby file and directory can act as a reader-writer lock: either one client handle may hold the lock in exclusive (writer) mode, or any number of client handles may

hold the lock in shared (reader) mode. Like the mutexes known to most programmers, locks are *advisory*. That is, they conflict only with other attempts to acquire the same lock: holding a lock called F neither is necessary to access the file F, nor prevents other clients from doing so. We rejected *mandatory* locks, which make locked objects inaccessible to clients not holding their locks:

- Chubby locks often protect resources implemented by other services, rather than just the file associated with the lock. To enforce mandatory locking in a meaningful way would have required us to make more extensive modification of these services.
- We did not wish to force users to shut down applications when they needed to access locked files for debugging or administrative purposes. In a complex system, it is harder to use the approach employed on most personal computers, where administrative software can break mandatory locks simply by instructing the user to shut down his applications or to reboot.
- Our developers perform error checking in the conventional way, by writing assertions such as "lock X is held", so they benefit little from mandatory checks. Buggy or malicious processes have many opportunities to corrupt data when locks are not held, so we find the extra guards provided by mandatory locking to be of no significant value.

In Chubby, acquiring a lock in either mode requires write permission so that an unprivileged reader cannot prevent a writer from making progress.

Locking is complex in distributed systems because communication is typically uncertain, and processes may fail independently. Thus, a process holding a lock *L* may issue a request *R*, but then fail. Another process may acquire *L* and perform some action before *R* arrives at its destination. If *R* later arrives, it may be acted on without the protection of *L*, and potentially on inconsistent data. The problem of receiving messages out of order has been well studied; solutions include *virtual time* [11], and *virtual synchrony* [1], which avoids the problem by ensuring that messages are processed in an order consistent with the observations of every participant.

It is costly to introduce sequence numbers into all the interactions in an existing complex system. Instead, Chubby provides a means by which sequence numbers can be introduced into only those interactions that make use of locks. At any time, a lock holder may request a *sequencer*, an opaque byte-string that describes the state of the lock immediately after acquisition. It contains the name of the lock, the mode in which it was acquired (exclusive or shared), and the lock generation number. The client passes the sequencer to servers (such as file servers) if it expects the operation to be protected by the lock. The recipient server is expected to test whether the sequencer is still valid and has the appropriate mode;

if not, it should reject the request. The validity of a sequencer can be checked against the server's Chubby cache or, if the server does not wish to maintain a session with Chubby, against the most recent sequencer that the server has observed. The sequencer mechanism requires only the addition of a string to affected messages, and is easily explained to our developers.

Although we find sequencers simple to use, important protocols evolve slowly. Chubby therefore provides an imperfect but easier mechanism to reduce the risk of delayed or re-ordered requests to servers that do not support sequencers. If a client releases a lock in the normal way, it is immediately available for other clients to claim, as one would expect. However, if a lock becomes free because the holder has failed or become inaccessible, the lock server will prevent other clients from claiming the lock for a period called the lock-delay. Clients may specify any lock-delay up to some bound, currently one minute; this limit prevents a faulty client from making a lock (and thus some resource) unavailable for an arbitrarily long time. While imperfect, the lock-delay protects unmodified servers and clients from everyday problems caused by message delays and restarts.

2.5 Events

Chubby clients may subscribe to a range of events when they create a handle. These events are delivered to the client asynchronously via an up-call from the Chubby library. Events include:

- file contents modified—often used to monitor the location of a service advertised via the file.
- child node added, removed, or modified—used to implement mirroring (§2.12). (In addition to allowing new files to be discovered, returning events for child nodes makes it possible to monitor ephemeral files without affecting their reference counts.)
- Chubby master failed over—warns clients that other events may have been lost, so data must be rescanned.
- a handle (and its lock) has become invalid—this typically suggests a communications problem.
- lock acquired—can be used to determine when a primary has been elected.
- conflicting lock request from another client—allows the caching of locks.

Events are delivered after the corresponding action has taken place. Thus, if a client is informed that file contents have changed, it is guaranteed to see the new data (or data that is yet more recent) if it subsequently reads the file.

The last two events mentioned are rarely used, and with hindsight could have been omitted. After primary election for example, clients typically need to communicate with the new primary, rather than simply know that a primary exists; thus, they wait for a file modifi-

cation event indicating that the new primary has written its address in a file. The conflicting lock event in theory permits clients to cache data held on other servers, using Chubby locks to maintain cache consistency. A notification of a conflicting lock request would tell a client to finish using data associated with the lock: it would finish pending operations, flush modifications to a home location, discard cached data, and release. So far, no one has adopted this style of use.

2.6 API

Clients see a Chubby handle as a pointer to an opaque structure that supports various operations. Handles are created only by open(), and destroyed with close().

Open() opens a named file or directory to produce a handle, analogous to a UNIX file descriptor. Only this call takes a node name; all others operate on handles.

The name is evaluated relative to an existing directory handle; the library provides a handle on "/" that is always valid. Directory handles avoid the difficulties of using a program-wide *current directory* in a multi-threaded program that contains many layers of abstraction [18].

The client indicates various options:

- how the handle will be used (reading; writing and locking; changing the ACL); the handle is created only if the client has the appropriate permissions.
- events that should be delivered (see §2.5).
- the lock-delay (§2.4).
- whether a new file or directory should (or must) be created. If a file is created, the caller may supply initial contents and initial ACL names. The return value indicates whether the file was in fact created.

Close() closes an open handle. Further use of the handle is not permitted. This call never fails. A related call Poison() causes outstanding and subsequent operations on the handle to fail without closing it; this allows a client to cancel Chubby calls made by other threads without fear of deallocating the memory being accessed by them.

The main calls that act on a handle are:

GetContentsAndStat() returns both the contents and meta-data of a file. The contents of a file are read atomically and in their entirety. We avoided partial reads and writes to discourage large files. A related call GetStat() returns just the meta-data, while ReadDir() returns the names and meta-data for the children of a directory.

SetContents() writes the contents of a file. Optionally, the client may provide a content generation number to allow the client to simulate compare-and-swap on a file; the contents are changed only if the generation number is current. The contents of a file are always written atomically and in their entirety. A related call SetACL() performs a similar operation on the ACL names associated with the node.

Delete() deletes the node if it has no children.

Acquire(), TryAcquire(), Release() acquire and release locks.

GetSequencer() returns a sequencer ($\S 2.4$) that describes any lock held by this handle.

SetSequencer () associates a sequencer with a handle. Subsequent operations on the handle fail if the sequencer is no longer valid.

CheckSequencer() checks whether a sequencer is valid (see §2.4).

Calls fail if the node has been deleted since the handle was created, even if the file has been subsequently recreated. That is, a handle is associated with an instance of a file, rather than with a file name. Chubby may apply access control checks on any call, but always checks Open () calls (see §2.3).

All the calls above take an *operation* parameter in addition to any others needed by the call itself. The operation parameter holds data and control information that may be associated with any call. In particular, via the operation parameter the client may:

- supply a callback to make the call asynchronous,
- · wait for the completion of such a call, and/or
- obtain extended error and diagnostic information.

Clients can use this API to perform primary election as follows: All potential primaries open the lock file and attempt to acquire the lock. One succeeds and becomes the primary, while the others act as replicas. The primary writes its identity into the lock file with SetContents() so that it can be found by clients and replicas, which read the file with GetContentsAndStat(), perhaps in response to a file-modification event (§2.5). Ideally, the primary obtains a sequencer with GetSequencer(), which it then passes to servers it communicates with; they should confirm with CheckSequencer() that it is still the primary. A lock-delay may be used with services that cannot check sequencers (§2.4).

2.7 Caching

To reduce read traffic, Chubby clients cache file data and node meta-data (including file absence) in a consistent, write-through cache held in memory. The cache is maintained by a lease mechanism described below, and kept consistent by invalidations sent by the master, which keeps a list of what each client may be caching. The protocol ensures that clients see either a consistent view of Chubby state, or an error.

When file data or meta-data is to be changed, the modification is blocked while the master sends invalidations for the data to every client that may have cached it; this mechanism sits on top of KeepAlive RPCs, discussed more fully in the next section. On receipt of an invalidation, a client flushes the invalidated state and acknowl-

edges by making its next KeepAlive call. The modification proceeds only after the server knows that each client has invalidated its cache, either because the client acknowledged the invalidation, or because the client allowed its cache lease to expire.

Only one round of invalidations is needed because the master treats the node as *uncachable* while cache invalidations remain unacknowledged. This approach allows reads always to be processed without delay; this is useful because reads greatly outnumber writes. An alternative would be to block calls that access the node during invalidation; this would make it less likely that over-eager clients will bombard the master with uncached accesses during invalidation, at the cost of occasional delays. If this were a problem, one could imagine adopting a hybrid scheme that switched tactics if overload were detected.

The caching protocol is simple: it invalidates cached data on a change, and never updates it. It would be just as simple to update rather than to invalidate, but update-only protocols can be arbitrarily inefficient; a client that accessed a file might receive updates indefinitely, causing an unbounded number of unnecessary updates.

Despite the overheads of providing strict consistency, we rejected weaker models because we felt that programmers would find them harder to use. Similarly, mechanisms such as virtual synchrony that require clients to exchange sequence numbers in all messages were considered inappropriate in an environment with diverse pre-existing communication protocols.

In addition to caching data and meta-data, Chubby clients cache open handles. Thus, if a client opens a file it has opened previously, only the first open() call necessarily leads to an RPC to the master. This caching is restricted in minor ways so that it never affects the semantics observed by the client: handles on ephemeral files cannot be held open if the application has closed them; and handles that permit locking can be reused, but cannot be used concurrently by multiple application handles. This last restriction exists because the client may use Close() or Poison() for their side-effect of cancelling outstanding Acquire() calls to the master.

Chubby's protocol permits clients to cache locks—that is, to hold locks longer than strictly necessary in the hope that they can be used again by the same client. An event informs a lock holder if another client has requested a conflicting lock, allowing the holder to release the lock just when it is needed elsewhere (see §2.5).

2.8 Sessions and KeepAlives

A Chubby session is a relationship between a Chubby cell and a Chubby client; it exists for some interval of time, and is maintained by periodic handshakes called KeepAlives. Unless a Chubby client informs the master

otherwise, the client's handles, locks, and cached data all remain valid provided its session remains valid. (However, the protocol for session maintenance may require the client to acknowledge a cache invalidation in order to maintain its session; see below.)

A client requests a new session on first contacting the master of a Chubby cell. It ends the session explicitly either when it terminates, or if the session has been idle (with no open handles and no calls for a minute).

Each session has an associated lease—an interval of time extending into the future during which the master guarantees not to terminate the session unilaterally. The end of this interval is called the session lease timeout. The master is free to advance this timeout further into the future, but may not move it backwards in time.

The master advances the lease timeout in three circumstances: on creation of the session, when a master fail-over occurs (see below), and when it responds to a KeepAlive RPC from the client. On receiving a KeepAlive, the master typically blocks the RPC (does not allow it to return) until the client's previous lease interval is close to expiring. The master later allows the RPC to return to the client, and thus informs the client of the new lease timeout. The master may extend the timeout by any amount. The default extension is 12s, but an overloaded master may use higher values to reduce the number of KeepAlive calls it must process. The client initiates a new KeepAlive immediately after receiving the previous reply. Thus, the client ensures that there is almost always a KeepAlive call blocked at the master.

As well as extending the client's lease, the KeepAlive reply is used to transmit events and cache invalidations back to the client. The master allows a KeepAlive to return early when an event or invalidation is to be delivered. Piggybacking events on KeepAlive replies ensures that clients cannot maintain a session without acknowledging cache invalidations, and causes all Chubby RPCs to flow from client to master. This simplifies the client, and allows the protocol to operate through firewalls that allow initiation of connections in only one direction.

The client maintains a local lease timeout that is a conservative approximation of the master's lease timeout. It differs from the master's lease timeout because the client must make conservative assumptions both of the time its KeepAlive reply spent in flight, and the rate at which the master's clock is advancing; to maintain consistency, we require that the server's clock advance no faster than a known constant factor faster than the client's.

If a client's local lease timeout expires, it becomes unsure whether the master has terminated its session. The client empties and disables its cache, and we say that its session is in *jeopardy*. The client waits a further interval called the grace period, 45s by default. If the client and master manage to exchange a successful KeepAlive be-

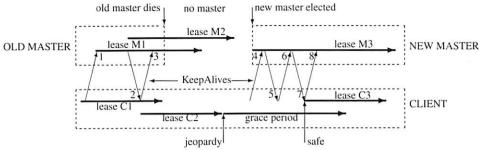


Figure 2: The role of the grace period in master fail-over

fore the end of the client's grace period, the client enables its cache once more. Otherwise, the client assumes that the session has expired. This is done so that Chubby API calls do not block indefinitely when a Chubby cell becomes inaccessible; calls return with an error if the grace period ends before communication is re-established.

The Chubby library can inform the application when the grace period begins via a *jeopardy* event. When the session is known to have survived the communications problem, a *safe* event tells the client to proceed; if the session times out instead, an *expired* event is sent. This information allows the application to quiesce itself when it is unsure of the status of its session, and to recover without restarting if the problem proves to be transient. This can be important in avoiding outages in services with large startup overhead.

If a client holds a handle H on a node and any operation on H fails because the associated session has expired, all subsequent operations on H (except Close()) and Poison()) will fail in the same way. Clients can use this to guarantee that network and server outages cause only a suffix of a sequence of operations to be lost, rather than an arbitrary subsequence, thus allowing complex changes to be marked as committed with a final write.

2.9 Fail-overs

When a master fails or otherwise loses mastership, it discards its in-memory state about sessions, handles, and locks. The authoritative timer for session leases runs at the master, so until a new master is elected the session lease timer is stopped; this is legal because it is equivalent to extending the client's lease. If a master election occurs quickly, clients can contact the new master before their local (approximate) lease timers expire. If the election takes a long time, clients flush their caches and wait for the grace period while trying to find the new master. Thus the grace period allows sessions to be maintained across fail-overs that exceed the normal lease timeout.

Figure 2 shows the sequence of events in a lengthy master fail-over event in which the client must use its grace period to preserve its session. Time increases from left to right, but times are not to scale. Client ses-

sion leases are shown as thick arrows both as viewed by both the old and new masters (M1-3, above) and the client (C1-3, below). Upward angled arrows indicate KeepAlive requests, and downward angled arrows their replies. The original master has session lease M1 for the client, while the client has a conservative approximation C1. The master commits to lease M2 before informing the client via KeepAlive reply 2; the client is able to extend its view of the lease C2. The master dies before replying to the next KeepAlive, and some time elapses before another master is elected. Eventually the client's approximation of its lease (C2) expires. The client then flushes its cache and starts a timer for the grace period.

During this period, the client cannot be sure whether its lease has expired at the master. It does not tear down its session, but it blocks all application calls on its API to prevent the application from observing inconsistent data. At the start of the grace period, the Chubby library sends a *jeopardy* event to the application to allow it to quiesce itself until it can be sure of the status of its session.

Eventually a new master election succeeds. The master initially uses a conservative approximation M3 of the session lease that its predecessor may have had for the client. The first KeepAlive request (4) from the client to the new master is rejected because it has the wrong master epoch number (described in detail below). The retried request (6) succeeds but typically does not extend the master lease further because M3 was conservative. However the reply (7) allows the client to extend its lease (C3) once more, and optionally inform the application that its session is no longer in jeopardy. Because the grace period was long enough to cover the interval between the end of lease C2 and the beginning of lease C3, the client saw nothing but a delay. Had the grace period been less than that interval, the client would have abandoned the session and reported the failure to the application.

Once a client has contacted the new master, the client library and master co-operate to provide the illusion to the application that no failure has occurred. To achieve this, the new master must reconstruct a conservative approximation of the in-memory state that the previous master had. It does this partly by reading data stored stably on disc (replicated via the normal database repli-

cation protocol), partly by obtaining state from clients, and partly by conservative assumptions. The database records each session, held lock, and ephemeral file.

A newly elected master proceeds:

- 1. It first picks a new client *epoch number*, which clients are required to present on every call. The master rejects calls from clients using older epoch numbers, and provides the new epoch number. This ensures that the new master will not respond to a very old packet that was sent to a previous master, even one running on the same machine.
- The new master may respond to master-location requests, but does not at first process incoming session-related operations.
- It builds in-memory data structures for sessions and locks that are recorded in the database. Session leases are extended to the maximum that the previous master may have been using.
- 4. The master now lets clients perform KeepAlives, but no other session-related operations.
- It emits a fail-over event to each session; this causes clients to flush their caches (because they may have missed invalidations), and to warn applications that other events may have been lost.
- 6. The master waits until each session acknowledges the fail-over event or lets its session expire.
- 7. The master allows all operations to proceed.
- 8. If a client uses a handle created prior to the fail-over (determined from the value of a sequence number in the handle), the master recreates the in-memory representation of the handle and honours the call. If such a recreated handle is closed, the master records it in memory so that it cannot be recreated in this master epoch; this ensures that a delayed or duplicated network packet cannot accidentally recreate a closed handle. A faulty client can recreate a closed handle in a future epoch, but this is harmless given that the client is already faulty.
- 9. After some interval (a minute, say), the master deletes ephemeral files that have no open file handles. Clients should refresh handles on ephemeral files during this interval after a fail-over. This mechanism has the unfortunate effect that ephemeral files may not disappear promptly if the last client on such a file loses its session during a fail-over.

Readers will be unsurprised to learn that the fail-over code, which is exercised far less often than other parts of the system, has been a rich source of interesting bugs.

2.10 Database implementation

The first version of Chubby used the replicated version of Berkeley DB [20] as its database. Berkeley DB provides B-trees that map byte-string keys to arbitrary byte-

string values. We installed a key comparison function that sorts first by the number of components in a path name; this allows nodes to by keyed by their path name, while keeping sibling nodes adjacent in the sort order. Because Chubby does not use path-based permissions, a single lookup in the database suffices for each file access.

Berkeley DB's uses a distributed consensus protocol to replicate its database logs over a set of servers. Once master leases were added, this matched the design of Chubby, which made implementation straightforward.

While Berkeley DB's B-tree code is widely-used and mature, the replication code was added recently, and has fewer users. Software maintainers must give priority to maintaining and improving their most popular product features. While Berkeley DB's maintainers solved the problems we had, we felt that use of the replication code exposed us to more risk than we wished to take. As a result, we have written a simple database using write ahead logging and snapshotting similar to the design of Birrell *et al.* [2]. As before, the database log is distributed among the replicas using a distributed consensus protocol. Chubby used few of the features of Berkeley DB, and so this rewrite allowed significant simplification of the system as a whole; for example, while we needed atomic operations, we did not need general transactions.

2.11 Backup

Every few hours, the master of each Chubby cell writes a snapshot of its database to a GFS file server [7] in a different building. The use of a separate building ensures both that the backup will survive building damage, and that the backups introduce no cyclic dependencies in the system; a GFS cell in the same building potentially might rely on the Chubby cell for electing its master.

Backups provide both disaster recovery and a means for initializing the database of a newly replaced replica without placing load on replicas that are in service.

2.12 Mirroring

Chubby allows a collection of files to be mirrored from one cell to another. Mirroring is fast because the files are small and the event mechanism (§2.5) informs the mirroring code immediately if a file is added, deleted, or modified. Provided there are no network problems, changes are reflected in dozens of mirrors world-wide in well under a second. If a mirror is unreachable, it remains unchanged until connectivity is restored. Updated files are then identified by comparing their checksums.

Mirroring is used most commonly to copy configuration files to various computing clusters distributed around the world. A special cell, named global, contains a subtree /ls/global/master that is mirrored to the

subtree /ls/cell/slave in every other Chubby cell. The global cell is special because its five replicas are located in widely-separated parts of the world, so it is almost always accessible from most of the organization.

Among the files mirrored from the global cell are Chubby's own access control lists, various files in which Chubby cells and other systems advertise their presence to our monitoring services, pointers to allow clients to locate large data sets such as Bigtable cells, and many configuration files for other systems.

3 Mechanisms for scaling

Chubby's clients are individual processes, so Chubby must handle more clients than one might expect; we have seen 90,000 clients communicating directly with a Chubby master—far more than the number of machines involved. Because there is just one master per cell, and its machine is identical to those of the clients, the clients can overwhelm the master by a huge margin. Thus, the most effective scaling techniques reduce communication with the master by a significant factor. Assuming the master has no serious performance bug, minor improvements in request processing at the master have little effect. We use several approaches:

- We can create an arbitrary number of Chubby cells; clients almost always use a nearby cell (found with DNS) to avoid reliance on remote machines. Our typical deployment uses one Chubby cell for a data centre of several thousand machines.
- The master may increase lease times from the default 12s up to around 60s when it is under heavy load, so it need process fewer KeepAlive RPCs. (KeepAlives are by far the dominant type of request (see 4.1), and failure to process them in time is the typical failure mode of an overloaded server; clients are largely insensitive to latency variation in other calls.)
- Chubby clients cache file data, meta-data, the absence of files, and open handles to reduce the number of calls they make on the server.
- We use protocol-conversion servers that translate the Chubby protocol into less-complex protocols such as DNS and others. We discuss some of these below.

Here we describe two familiar mechanisms, proxies and partitioning, that we expect will allow Chubby to scale further. We do not yet use them in production, but they are designed, and may be used soon. We have no present need to consider scaling beyond a factor of five: First, there are limits on the number of machines one would wish to put in a data centre or make reliant on a single instance of a service. Second, because we use similar machines for Chubby clients and servers, hardware improvements that increase the number of clients per machine also increase the capacity of each server.

3.1 Proxies

Chubby's protocol can be proxied (using the same protocol on both sides) by trusted processes that pass requests from other clients to a Chubby cell. A proxy can reduce server load by handling both KeepAlive and read requests; it cannot reduce write traffic, which passes through the proxy's cache. But even with aggressive client caching, write traffic constitutes much less than one percent of Chubby's normal workload (see §4.1), so proxies allow a significant increase in the number of clients. If a proxy handles N_{proxy} clients, KeepAlive traffic is reduced by a factor of N_{proxy} , which might be 10 thousand or more. A proxy cache can reduce read traffic by at most the mean amount of read-sharing—a factor of around 10 (§4.1). But because reads constitute under 10% of Chubby's load at present, the saving in KeepAlive traffic is by far the more important effect.

Proxies add an additional RPC to writes and first-time reads. One might expect proxies to make the cell temporarily unavailable at least twice as often as before, because each proxied client depends on two machines that may fail: its proxy and the Chubby master.

Alert readers will notice that the fail-over strategy described in Section 2.9, is not ideal for proxies. We discuss this problem in Section 4.4.

3.2 Partitioning

As mentioned in Section 2.3, Chubby's interface was chosen so that the name space of a cell could be partitioned between servers. Although we have not yet needed it, the code can partition the name space by directory. If enabled, a Chubby cell would be composed of N partitions, each of which has a set of replicas and a master. Every node D/C in directory D would be stored on the partition $P(D/C) = \mathtt{hash}(D) \bmod N$. Note that the meta-data for D may be stored on a different partition $P(D) = \mathtt{hash}(D') \bmod N$, where D' is the parent of D.

Partitioning is intended to enable large Chubby cells with little communication between the partitions. Although Chubby lacks hard links, directory modified-times, and cross-directory rename operations, a few operations still require cross-partition communication:

- ACLs are themselves files, so one partition may use another for permissions checks. However, ACL files are readily cached; only Open() and Delete() calls require ACL checks; and most clients read publicly accessible files that require no ACL.
- When a directory is deleted, a cross-partition call may be needed to ensure that the directory is empty.

Because each partition handles most calls independently of the others, we expect this communication to have only a modest impact on performance or availability. Unless the number of partitions N is large, one would expect that each client would contact the majority of the partitions. Thus, partitioning reduces read and write traffic on any given partition by a factor of N but does not necessarily reduce KeepAlive traffic. Should it be necessary for Chubby to handle more clients, our strategy involves a combination of proxies and partitioning.

4 Use, surprises and design errors

4.1 Use and behaviour

The following table gives statistics taken as a snapshot of a Chubby cell; the RPC rate was a seen over a ten-minute period. The numbers are typical of cells in Google.

time since last fail-over	18 days
fail-over duration	14s
active clients (direct)	22k
additional proxied clients	32k
files open	12k
naming-related	60%
client-is-caching-file entries	230k
distinct files cached	24k
names negatively cached	32k
exclusive locks	1k
shared locks	0
stored directories	8k
ephemeral	0.1%
stored files	22k
0-1k bytes	90%
1k-10k bytes	10%
> 10k bytes	0.2%
naming-related	46%
mirrored ACLs & config info	27%
GFS and Bigtable meta-data	11%
ephemeral	3%
RPC rate	1-2k/s
KeepAlive	93%
GetStat	2%
Open	1%
CreateSession	1%
GetContentsAndStat	0.4%
SetContents	680ppm
Acquire	31ppm

Several things can be seen:

- Many files are used for naming; see §4.3.
- Configuration, access control, and meta-data files (analogous to file system super-blocks) are common.
- Negative caching is significant.
- 230k/24k≈10 clients use each cached file, on average.
- Few clients hold locks, and shared locks are rare; this
 is consistent with locking being used for primary election and partitioning data among replicas.

 RPC traffic is dominated by session KeepAlives; there are a few reads (which are cache misses); there are very few writes or lock acquisitions.

Now we briefly describe the typical causes of outages in our cells. If we assume (optimistically) that a cell is "up" if it has a master that is willing to serve, on a sample of our cells we recorded 61 outages over a period of a few weeks, amounting to 700 cell-days of data in total. We excluded outages due to maintenance that shut down the data centre. All other causes are included: network congestion, maintenance, overload, and errors due to operators, software, and hardware. Most outages were 15s or less, and 52 were under 30s; most of our applications are not affected significantly by Chubby outages under 30s. The remaining nine outages were caused by network maintenance (4), suspected network connectivity problems (2), software errors (2), and overload (1).

In a few dozen cell-years of operation, we have lost data on six occasions, due to database software errors (4) and operator error (2); none involved hardware error. Ironically, the operational errors involved upgrades to avoid the software errors. We have twice corrected corruptions caused by software in non-master replicas.

Chubby's data fits in RAM, so most operations are cheap. Mean request latency at our production servers is consistently a small fraction of a millisecond regardless of cell load until the cell approaches overload, when latency increases dramatically and sessions are dropped. Overload typically occurs when many sessions (> 90,000) are active, but can result from exceptional conditions: when clients made millions of read requests simultaneously (described in Section 4.3), and when a mistake in the client library disabled caching for some reads, resulting in tens of thousands of requests per second. Because most RPCs are KeepAlives, the server can maintain a low mean request latency with many active clients by increasing the session lease period (see §3). Group commit reduces the effective work done per request when bursts of writes arrive, but this is rare.

RPC read latencies measured at the client are limited by the RPC system and network; they are under 1ms for a local cell, but 250ms between antipodes. Writes (which include lock operations) are delayed a further 5-10ms by the database log update, but by up to tens of seconds if a recently-failed client cached the file. Even this variability in write latency has little effect on the mean request latency at the server because writes are so infrequent.

Clients are fairly insensitive to latency variation provided sessions are not dropped. At one point, we added artificial delays in <code>Open()</code> to curb abusive clients (see §4.5); developers noticed only when delays exceeded ten seconds *and* were applied repeatedly. We have found that the key to scaling Chubby is not server performance; reducing communication to the server can have far greater

impact. No significant effort has been applied to tuning read/write server code paths; we checked that no egregious bugs were present, then focused on the scaling mechanisms that could be more effective. On the other hand, developers do notice if a performance bug affects the local Chubby cache, which a client may read thousands of times per second.

4.2 Java clients

Google's infrastructure is mostly in C++, but a growing number of systems are being written in Java [8]. This trend presented an unanticipated problem for Chubby, which has a complex client protocol and a non-trivial client-side library.

Java encourages portability of entire applications at the expense of incremental adoption by making it somewhat irksome to link against other languages. The usual Java mechanism for accessing non-native libraries is JNI [15], but it is regarded as slow and cumbersome. Our Java programmers so dislike JNI that to avoid its use they prefer to translate large libraries into Java, and commit to supporting them.

Chubby's C++ client library is 7000 lines (comparable with the server), and the client protocol is delicate. To maintain the library in Java would require care and expense, while an implementation without caching would burden the Chubby servers. Thus our Java users run copies of a protocol-conversion server that exports a simple RPC protocol that corresponds closely to Chubby's client API. Even with hindsight, it is not obvious how we might have avoided the cost of writing, running and maintaining this additional server.

4.3 Use as a name service

Even though Chubby was designed as a lock service, we found that its most popular use was as a name server.

Caching within the normal Internet naming system, the DNS, is based on time. DNS entries have a *time-to-live* (TTL), and DNS data are discarded when they have not been refreshed within that period. Usually it is straightforward to pick a suitable TTL value, but if prompt replacement of failed services is desired, the TTL can become small enough to overload the DNS servers.

For example, it is common for our developers to run jobs involving thousands of processes, and for each process to communicate with every other, leading to a quadratic number of DNS lookups. We might wish to use a TTL of 60s; this would allow misbehaving clients to be replaced without excessive delay and is not considered an unreasonably short replacement time in our environment. In that case, to maintain the DNS caches

of a single job as small as 3 thousand clients would require 150 thousand lookups per second. (For comparison, a 2-CPU 2.6GHz Xeon DNS server might handle 50 thousand requests per second.) Larger jobs create worse problems, and several jobs many be running at once. The variability in our DNS load had been a serious problem for Google before Chubby was introduced.

In contrast, Chubby's caching uses explicit invalidations so a constant rate of session KeepAlive requests can maintain an arbitrary number of cache entries indefinitely at a client, in the absence of changes. A 2-CPU 2.6GHz Xeon Chubby master has been seen to handle 90 thousand clients communicating directly with it (no proxies); the clients included large jobs with communication patterns of the kind described above. The ability to provide swift name updates without polling each name individually is so appealing that Chubby now provides name service for most of the company's systems.

Although Chubby's caching allows a single cell to sustain a large number of clients, load spikes can still be a problem. When we first deployed the Chubby-based name service, starting a 3 thousand process job (thus generating 9 million requests) could bring the Chubby master to its knees. To resolve this problem, we chose to group name entries into batches so that a single lookup would return and cache the name mappings for a large number (typically 100) of related processes within a job.

The caching semantics provided by Chubby are more precise than those needed by a name service; name resolution requires only timely notification rather than full consistency. As a result, there was an opportunity for reducing the load on Chubby by introducing a simple protocol-conversion server designed specifically for name lookups. Had we foreseen the use of Chubby as a name service, we might have chosen to implement full proxies sooner than we did in order to avoid the need for this simple, but nevertheless additional server.

One further protocol-conversion server exists: the Chubby DNS server. This makes the naming data stored within Chubby available to DNS clients. This server is important both for easing the transition from DNS names to Chubby names, and to accommodate existing applications that cannot be converted easily, such as browsers.

4.4 Problems with fail-over

The original design for master fail-over (§2.9) requires the master to write new sessions to the database as they are created. In the Berkeley DB version of the lock server, the overhead of creating sessions became a problem when many processes were started at once. To avoid overload, the server was modified to store a session in the database not when it was first created, but instead when it attempted its first modification, lock acquisition, or open

of an ephemeral file. In addition, active sessions were recorded in the database with some probability on each KeepAlive. Thus, the writes for read-only sessions were spread out in time.

Though it was necessary to avoid overload, this optimization has the undesirable effect that young read-only sessions may not be recorded in the database, and so may be discarded if a fail-over occurs. Although such sessions hold no locks, this is unsafe; if all the recorded sessions were to check in with the new master before the leases of discarded sessions expired, the discarded sessions could then read stale data for a while. This is rare in practice; in a large system it is almost certain that some session will fail to check in, and thus force the new master to await the maximum lease time anyway. Nevertheless, we have modified the fail-over design both to avoid this effect, and to avoid a complication that the current scheme introduces to proxies.

Under the new design, we avoid recording sessions in the database at all, and instead recreate them in the same way that the master currently recreates handles ($\S2.9,\P8$). A new master must now wait a full worst-case lease timeout before allowing operations to proceed, since it cannot know whether all sessions have checked in ($\S2.9,\P6$). Again, this has little effect in practice because it is likely that not all sessions will check in.

Once sessions can be recreated without on-disc state, proxy servers can manage sessions that the master is not aware of. An extra operation available only to proxies allows them to change the session that locks are associated with. This permits one proxy to take over a client from another when a proxy fails. The only further change needed at the master is a guarantee not to relinquish locks or ephemeral file handles associated with proxy sessions until a new proxy has had a chance to claim them.

4.5 Abusive clients

Google's project teams are free to set up their own Chubby cells, but doing so adds to their maintenance burden, and consumes additional hardware resources. Many services therefore use shared Chubby cells, which makes it important to isolate clients from the misbehaviour of others. Chubby is intended to operate within a single company, and so malicious denial-of-service attacks against it are rare. However, mistakes, misunderstandings, and the differing expectations of our developers lead to effects that are similar to attacks.

Some of our remedies are heavy-handed. For example, we review the ways project teams plan to use Chubby, and deny access to the shared Chubby name space until review is satisfactory. A problem with this approach is that developers are often unable to predict how their services will be used in the future, and how use will grow.

Readers will note the irony of our own failure to predict how Chubby itself would be used.

The most important aspect of our review is to determine whether use of any of Chubby's resources (RPC rate, disc space, number of files) grows linearly (or worse) with number of users or amount of data processed by the project. Any linear growth must be mitigated by a compensating parameter that can be adjusted to reduce the load on Chubby to reasonable bounds. Nevertheless our early reviews were not thorough enough.

A related problem is the lack of performance advice in most software documentation. A module written by one team may be reused a year later by another team with disastrous results. It is sometimes hard to explain to interface designers that they must change their interfaces not because they are bad, but because other developers may be less aware of the cost of an RPC.

Below we list some problem cases we encountered.

Lack of aggressive caching Originally, we did not appreciate the critical need to cache the absence of files, nor to reuse open file handles. Despite attempts at education, our developers regularly write loops that retry indefinitely when a file is not present, or poll a file by opening it and closing it repeatedly when one might expect they would open the file just once.

At first we countered these retry-loops by introducing exponentially-increasing delays when an application made many attempts to Open() the same file over a short period. In some cases this exposed bugs that developers acknowledged, but often it required us to spend yet more time on education. In the end it was easier to make repeated Open() calls cheap.

Lack of quotas Chubby was never intended to be used as a storage system for large amounts of data, and so it has no storage quotas. In hindsight, this was naïve.

One of Google's projects wrote a module to keep track of data uploads, storing some meta-data in Chubby. Such uploads occurred rarely and were limited to a small set of people, so the space was bounded. However, two other services started using the same module as a means for tracking uploads from a wider population of users. Inevitably, these services grew until the use of Chubby was extreme: a single 1.5MByte file was being rewritten in its entirety on each user action, and the overall space used by the service exceeded the space needs of all other Chubby clients combined.

We introduced a limit on file size (256kBytes), and encouraged the services to migrate to more appropriate storage systems. But it is difficult to make significant changes to production systems maintained by busy people—it took approximately a year for the data to be migrated elsewhere.

Publish/subscribe There have been several attempts to use Chubby's event mechanism as a publish/subscribe

system in the style of Zephyr [6]. Chubby's heavyweight guarantees and its use of invalidation rather than update in maintaining cache consistency make it a slow and inefficient for all but the most trivial publish/subscribe examples. Fortunately, all such uses have been caught before the cost of redesigning the application was too large.

4.6 Lessons learned

Here we list lessons, and miscellaneous design changes we might make if we have the opportunity:

Developers rarely consider availability We find that our developers rarely think about failure probabilities, and are inclined to treat a service like Chubby as though it were always available. For example, our developers once built a system employing hundred of machines that initiated recovery procedures taking tens of minutes when Chubby elected a new master. This magnified the consequences of a single failure by a factor of a hundred both in time *and* the number of machines affected. We would prefer developers to plan for short Chubby outages, so that such an event has little or no affect on their applications. This is one of the arguments for coarsegrained locking, discussed in Section 2.1.

Developers also fail to appreciate the difference between a service being up, and that service being available to their applications. For example, the global Chubby cell (see §2.12), is almost always up because it is rare for more than two geographically distant data centres to be down simultaneously. However, its *observed availability for a given client* is usually lower than the observed availability of the client's local Chubby cell. First, the local cell is less likely to be partitioned from the client, and second, although the local cell may be down often due to maintenance, the same maintenance affects the client directly, so Chubby's unavailability is not observed by the client.

Our API choices can also affect the way developers chose to handle Chubby outages. For example, Chubby provides an event that allows clients to detect when a master fail-over has taken place. The intent was for clients to check for possible changes, as other events may have been lost. Unfortunately, many developers chose to crash their applications on receiving this event, thus decreasing the availability of their systems substantially. We might have done better to send redundant "file change" events instead, or even to ensure that no events were lost during a fail-over.

At present we use three mechanisms to prevent developers from being over-optimistic about Chubby availability, especially that of the global cell. First, as previously mentioned (§4.5), we review how project teams plan to use Chubby, and advise them against techniques that would tie their availability too closely to Chubby's.

Second, we now supply libraries that perform some highlevel tasks so that developers are automatically isolated from Chubby outages. Third, we use the post-mortem of each Chubby outage as a means not only of eliminating bugs in Chubby and our operational procedures, but of reducing the sensitivity of applications to Chubby's availability—both can lead to better availability of our systems overall.

Fine-grained locking could be ignored At the end of Section 2.1 we sketched a design for a server that clients could run to provide fine-grained locking. It is perhaps a surprise that so far we have not needed to write such a server; our developers typically find that to optimize their applications, they must remove unnecessary communication, and that often means finding a way to use coarse-grained locking.

Poor API choices have unexpected affects For the most part, our API has evolved well, but one mistake stands out. Our means for cancelling long-running calls are the close() and Poison() RPCs, which also discard the server state for the handle. This prevents handles that can acquire locks from being shared, for example, by multiple threads. We may add a Cancel() RPC to allow more sharing of open handles.

RPC use affects transport protocols KeepAlives are used both for refreshing the client's session lease, and for passing events and cache invalidations from the master to the client. This design has the automatic and desirable consequence that a client cannot refresh its session lease without acknowledging cache invalidations.

This would seem ideal, except that it introduced a tension in our choice of protocol. TCP's back off policies pay no attention to higher-level timeouts such as Chubby leases, so TCP-based KeepAlives led to many lost sessions at times of high network congestion. We were forced to send KeepAlive RPCs via UDP rather than TCP; UDP has no congestion avoidance mechanisms, so we would prefer to use UDP only when high-level time-bounds must be met.

We may augment the protocol with an additional TCP-based <code>GetEvent()</code> RPC which would be used to communicate events and invalidations in the normal case, used in the same way KeepAlives. The KeepAlive reply would still contain a list of unacknowledged events so that events must eventually be acknowledged.

5 Comparison with related work

Chubby is based on well-established ideas. Chubby's cache design is derived from work on distributed file systems [10]. Its sessions and cache tokens are similar in behaviour to those in Echo [17]; sessions reduce the overhead of leases [9] in the V system. The idea of exposing a general-purpose lock service is found in VMS [23],

though that system initially used a special-purpose highspeed interconnect that permitted low-latency interactions. Like its caching model, Chubby's API is based on a file-system model, including the idea that a filesystem-like name space is convenient for more than just files [18, 21, 22].

Chubby differs from a distributed file system such as Echo or AFS [10] in its performance and storage aspirations: Clients do not read, write, or store large amounts of data, and they do not expect high throughput or even low-latency unless the data is cached. They do expect consistency, availability, and reliability, but these attributes are easier to achieve when performance is less important. Because Chubby's database is small, we are able to store many copies of it on-line (typically five replicas and a few backups). We take full backups multiple times per day, and via checksums of the database state, we compare replicas with one another every few hours. The weakening of the normal file system performance and storage requirements allows us to serve tens of thousands of clients from a single Chubby master. By providing a central point where many clients can share information and co-ordinate activities, we have solved a class of problems faced by our system developers.

The large number of file systems and lock servers described in the literature prevents an exhaustive comparison, so we provide details on one: we chose to compare with Boxwood's lock server [16] because it was designed recently, it too is designed to run in a loosely-coupled environment, and yet its design differs in various ways from Chubby, some interesting and some incidental.

Chubby implements locks, a reliable small-file storage system, and a session/lease mechanism in a single service. In contrast, Boxwood separates these into three: a lock service, a Paxos service (a reliable repository for state), and a failure detection service respectively. The Boxwood system itself uses these three components together, but another system could use these building blocks independently. We suspect that this difference in design stems from a difference in target audience. Chubby was intended for a diverse audience and application mix; its users range from experts who create new distributed systems, to novices who write administration scripts. For our environment, a large-scale shared service with a familiar API seemed attractive. In contrast, Boxwood provides a toolkit that (to our eyes, at least) is appropriate for a smaller number of more sophisticated developers working on projects that may share code but need not be used together.

In many cases, Chubby provides a higher-level interface than Boxwood. For example, Chubby combines the lock and file names spaces, while Boxwood's lock names are simple byte sequences. Chubby clients cache file state by default; a client of Boxwood's Paxos service

could implement caching via the lock service, but would probably use the caching provided by Boxwood itself.

The two systems have markedly different default parameters, chosen for different expectations: Each Boxwood failure detector is contacted by each client every 200ms with a timeout of 1s; Chubby's default lease time is 12s and KeepAlives are exchanged every 7s. Boxwood's subcomponents use two or three replicas to achieve availability, while we typically use five replicas per cell. However, these choices alone do not suggest a deep design difference, but rather an indication of how parameters in such systems must be adjusted to accommodate more client machines, or the uncertainties of racks shared with other projects.

A more interesting difference is the introduction of Chubby's grace period, which Boxwood lacks. (Recall that the grace period allows clients to ride out long Chubby master outages without losing sessions or locks. Boxwood's "grace period" is the equivalent of Chubby's "session lease", a different concept.) Again, this difference is the result of differing expectations about scale and failure probability in the two systems. Although master fail-overs are rare, a lost Chubby lock is expensive for clients.

Finally, locks in the two systems are intended for different purposes. Chubby locks are heavier-weight, and need sequencers to allow externals resources to be protected safely, while Boxwood locks are lighter-weight, and intended primarily for use within Boxwood.

6 Summary

Chubby is a distributed lock service intended for coarsegrained synchronization of activities within Google's distributed systems; it has found wider use as a name service and repository for configuration information.

Its design is based on well-known ideas that have meshed well: distributed consensus among a few replicas for fault tolerance, consistent client-side caching to reduce server load while retaining simple semantics, timely notification of updates, and a familiar file system interface. We use caching, protocol-conversion servers, and simple load adaptation to allow it scale to tens of thousands of client processes per Chubby instance. We expect to scale it further via proxies and partitioning.

Chubby has become Google's primary internal name service; it is a common rendezvous mechanism for systems such as MapReduce [4]; the storage systems GFS and Bigtable use Chubby to elect a primary from redundant replicas; and it is a standard repository for files that require high availability, such as access control lists.

Acknowledgments

Many contributed to the Chubby system: Sharon Perl wrote the replication layer on Berkeley DB; Tushar Chandra and Robert Griesemer wrote the replicated database that replaced Berkeley DB; Ramsey Haddad connected the API to Google's file system interface; Dave Presotto, Sean Owen, Doug Zongker and Praveen Tamara wrote the Chubby DNS, Java, and naming protocol-converters, and the full Chubby proxy respectively; Vadim Furman added the caching of open handles and file-absence; Rob Pike, Sean Quinlan and Sanjay Ghemawat gave valuable design advice; and many Google developers uncovered early weaknesses.

References

- [1] BIRMAN, K. P., AND JOSEPH, T. A. Exploiting virtual synchrony in distributed systems. In 11th SOSP (1987), pp. 123-138.
- [2] BIRRELL, A., JONES, M. B., AND WOBBER, E. A simple and efficient implementation for small databases. In 11th SOSP (1987), pp. 149-154.
- [3] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A distributed structured data storage system. In 7th OSDI (2006).
- [4] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In 6th OSDI (2004), pp. 137-150.
- [5] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. J. ACM 32, 2 (April 1985), 374-382.
- [6] FRENCH, R. S., AND KOHL, J. T. The Zephyr Programmer's Manual. MIT Project Athena, Apr. 1989.
- [7] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In 19th SOSP (Dec. 2003), pp. 29–43.
- [8] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. Java Language Spec. (2nd Ed.). Addison-Wesley, 2000.
- [9] GRAY, C. G., AND CHERITON, D. R. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In 12th SOSP (1989), pp. 202-210.
- [10] HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. Scale and performance in a distributed file system. ACM TOCS 6, 1 (Feb. 1988), 51-81.
- [11] JEFFERSON, D. Virtual time. ACM TOPLAS, 3 (1985), 404-425.
- [12] LAMPORT, L. The part-time parliament. ACM TOCS 16, 2 (1998), 133–169.
- [13] LAMPORT, L. Paxos made simple. ACM SIGACT News 32, 4 (2001), 18-25.

- [14] LAMPSON, B. W. How to build a highly available system using consensus. In Distributed Algorithms, vol. 1151 of LNCS. Springer-Verlag, 1996, pp. 1-17.
- [15] LIANG, S. Java Native Interface: Programmer's Guide and Reference. Addison-Wesley, 1999.
- [16] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In 6th OSDI (2004), pp. 105-120.
- [17] MANN, T., BIRRELL, A., HISGEN, A., JERIAN, C., AND SWART, G. A coherent distributed file cache with directory write-behind. TOCS 12, 2 (1994), 123-164.
- [18] McJones, P., and Swart, G. Evolving the UNIX system interface to support multithreaded programs. Tech. Rep. 21, DEC SRC, 1987.
- [19] OKI, B., AND LISKOV, B. Viewstamped replication: A general primary copy method to support highly-available distributed systems. In ACM PODC (1988).
- [20] OLSON, M. A., BOSTIC, K., AND SELTZER, M. Berkeley DB. In USENIX (June 1999), pp. 183-192.
- [21] PIKE, R., PRESOTTO, D. L., DORWARD, S., FLAN-DRENA, B., THOMPSON, K., TRICKEY, H., AND WIN-TERBOTTOM, P. Plan 9 from Bell Labs. Computing Systems 8, 2 (1995), 221-254.
- [22] RITCHIE, D. M., AND THOMPSON, K. The UNIX timesharing system. CACM 17, 7 (1974), 365–375.
- [23] SNAMAN, JR., W. E., AND THIEL, D. W. VAX/VMS distributed lock manager. Digital Technical Journal 1, 5 (Sept. 1987), 29-44.
- [24] YIN, J., MARTIN, J.-P., VENKATARAMANI, A., ALVISI, L., AND DAHLIN, M. Separating agreement from execution for byzantine fault tolerant services. In 19th SOSP (2003), pp. 253-267.

Experiences Building PlanetLab

Larry Peterson, Andy Bavier, Marc E. Fiuczynski, Steve Muir

Department of Computer Science

Princeton University

Abstract. This paper reports our experiences building PlanetLab over the last four years. It identifies the requirements that shaped PlanetLab, explains the design decisions that resulted from resolving conflicts among these requirements, and reports our experience implementing and supporting the system. Due in large part to the nature of the "PlanetLab experiment," the discussion focuses on synthesis rather than new techniques, balancing system-wide considerations rather than improving performance along a single dimension, and learning from feedback from a live system rather than controlled experiments using synthetic workloads.

1 Introduction

PlanetLab is a global platform for deploying and evaluating network services [21, 3]. In many ways, it has been an unexpected success. It was launched in mid-2002 with 100 machines distributed to 40 sites, but today includes 700 nodes spanning 336 sites and 35 countries. It currently hosts 2500 researchers affiliated with 600 projects. It has been used to evaluate a diverse set of planetary-scale network services, including content distribution [33, 8, 24], anycast [35, 9], DHTs [26], robust DNS [20, 25], large-file distribution [19, 1], measurement and analysis [30], anomaly and fault diagnosis [36], and event notification [23]. It supports the design and evaluation of dozens of long-running services that transport an aggregate of 3-4TB of data every day, satisfying tens of millions of requests involving roughly one million unique clients and servers.

To deliver this utility, PlanetLab innovates along two main dimensions:

- Novel management architecture. PlanetLab administers nodes owned by hundreds of organizations, which agree to allow a worldwide community of researchers—most complete strangers—to access their machines. PlanetLab must manage a complex relationship between node owners and users.
- Novel usage model. Each PlanetLab node should gracefully degrade in performance as the number of users grows. This gives the PlanetLab community an incentive to work together to make best use of its shared resources.

In both cases, the contribution is not a new mechanism or algorithm, but rather a synthesis (and full exploitation) of carefully selected ideas to produce a fundamentally new system.

Moreover, the process by which we designed the system is interesting in its own right:

- Experience-driven design. PlanetLab's design evolved incrementally based on experience gained from supporting a live user community. This is in contrast to most research systems that are designed and evaluated under controlled conditions, contained within a single organization, and evaluated using synthetic workloads.
- Conflict-driven design. The design decisions that shaped PlanetLab were responses to conflicting requirements. The result is a comprehensive architecture based more on balancing global considerations than improving performance along a single dimension, and on real-world requirements that do not always lend themselves to quantifiable metrics.

One could view this as a new model of system design, but of course it isn't [6, 27].

This paper identifies the requirements that shaped the system, explains the design decisions that resulted from resolving conflicts among these requirements, and reports our experience building and supporting the system. A side-effect of the discussion is a fairly complete overview of PlanetLab's current architecture, but the primary goal is to describe the design decisions that went into building PlanetLab, and to report the lessons we have learned in the process. For a comprehensive definition of the PlanetLab architecture, the reader is referred to [22].

2 Background

This section identifies the requirements we understood at the time PlanetLab was first conceived, and sketches the high-level design proposed at that time. The discussion includes a summary of the three main challenges we have faced, all of which can be traced to tensions between the requirements. The section concludes by looking at the relationship between PlanetLab and similar systems.

2.1 Requirements

PlanetLab's design was guided by five major requirements that correspond to objectives we hoped to achieve as well as constraints we had to live with. Although we recognized all of these requirements up-front, the following discussion articulates them with the benefit of hindsight.

(R1) It must provide a global platform that supports both short-term experiments and long-running services. Unlike previous testbeds, a revolutionary goal of Planet-Lab was that it support experimental services that could run continuously and support a real client workload. This implied that multiple services be able to run concurrently since a batch-scheduled facility is not conducive to a 24×7 workload. Moreover, these services (experiments) should be isolated from each other so that one service does not unduly interfere with another.

(R2) It must be available immediately, even though no one knows for sure what "it" is. PlanetLab faced a dilemma: it was designed to support research in broad-coverage network services, yet its management (control) plane is itself such a service. It was necessary to deploy PlanetLab and start gaining experience with network services before we fully understood what services would be needed to manage it. As a consequence, PlanetLab had to be designed with explicit support for evolution. Moreover, to get people to use PlanetLab—so we could learn from it—it had to be as familiar as possible; researchers are not likely to change their programming environment to use a new facility.

(R3) We must convince sites to host nodes running code written by unknown researchers from other organizations. PlanetLab takes advantage of nodes contributed by research organizations around the world. These nodes, in turn, host services on behalf of users from other research organizations. The individual users are unknown to the node owners, and to make matters worse, the services they deploy often send potentially disruptive packets into the Internet. That sites own and host nodes, but trust PlanetLab to administer them, is unprecedented at the scale PlanetLab operates. As a consequence, we must correctly manage the trust relationships so that the risks to each site are less than the benefits they derive.

(R4) Sustaining growth depends on support for autonomy and decentralized control. PlanetLab is a world-wide platform constructed from components owned by many autonomous organizations. Each organization must retain some amount of control over how their resources are used, and PlanetLab as a whole must give geographic regions and other communities as much autonomy as possible in defining and managing the system. Generally, sustaining such a system requires minimizing centralized control.

(R5) It must scale to support many users with minimal resources. While a commercial variant of PlanetLab might have cost recovery mechanisms to provide resource guarantees to each of its users, PlanetLab must operate in an under-provisioned environment. This means conservative allocation strategies are not practical, and it is necessary to promote efficient resource sharing. This includes both physical resources (e.g., cycles, bandwidth, and memory) and logical resources (e.g., IP addresses).

Note that while the rest of this paper discusses the many tensions between these requirements, two of them are quite synergistic. The requirement that we evolve PlanetLab (R2) and the need for decentralized control (R4) both point to the value of factoring PlanetLab's management architecture into a set of building block components with well-defined interfaces. A major challenge of building PlanetLab was to understand exactly what these pieces should be.

To this end, PlanetLab originally adopted an organizing principle called *unbundled management*, which argued that the services used to manage PlanetLab should themselves be deployed like any other service, rather than bundled with the core system. The case for unbundled management has three arguments: (1) to allow the system to more easily evolve; (2) to permit third-party developers to build alternative services, enabling a software bazaar, rather than rely on a single development team with limited resources and creativity; and (3) to permit decentralized control over PlanetLab resources, and ultimately, over its evolution.

2.2 Initial Design

PlanetLab supports the required usage model through distributed virtualization—each service runs in a slice of PlanetLab's global resources. Multiple slices run concurrently on PlanetLab, where slices act as network-wide containers that isolate services from each other. Slices were expected to enforce two kinds of isolation: resource isolation and security isolation, the former concerned with minimizing performance interference and the latter concerned with eliminating namespace interference.

At a high-level, PlanetLab consists of a centralized front-end, called PlanetLab Central (PLC), that remotely manages a set of nodes. Each node runs a *node manager* (NM) that establishes and controls *virtual machines* (VM) on that node. We assume an underlying *virtual machine monitor* (VMM) implements the VMs. Users create slices through operations available on PLC, which results in PLC contacting the NM on each node to create a local VM. A set of such VMs defines the slice.

We initially elected to use a Linux-based VMM due to Linux's high mind-share [3]. Linux is augmented with Vservers [16] to provide security isolation and a set of schedulers to provide resource isolation.

2.3 Design Challenges

Like many real systems, what makes PlanetLab interesting to study—and challenging to build—is how it deals with the constraints of reality and conflicts among requirements. Here, we summarize the three main challenges; subsequent sections address each in more detail.

First, unbundled management is a powerful design principle for evolving a system, but we did not fully understand what it entailed nor how it would be shaped by other aspects of the system. Defining PlanetLab's management architecture—and in particular, deciding how to factor management functionality into a set of independent pieces—involved resolving three main conflicts:

- minimizing centralized components (R4) yet maintaining the necessary trust assumptions (R3);
- balancing the need for slices to acquire the resources they need (R1) yet coping with scarce resources (R5);
- isolating slices from each other (R1) yet allowing some slices to manage other slices (R2).

Section 3 discusses our experiences evolving PlanetLab's management architecture.

Second, resource allocation is a significant challenge for any system, and this is especially true for PlanetLab, where the requirement for isolation (R1) is in conflict with the reality of limited resources (R5). Part of our approach to this situation is embodied in the management structure described in Section 3, but it is also addressed in how scheduling and allocation decisions are made on a per-node basis. Section 4 reports our experience balancing isolation against efficient resource usage.

Third, we must maintain a stable system on behalf of the user community (R1) and yet evolve the platform to provide long-term viability and sustainability (R2). Section 5 reports our operational experiences with Planet-Lab, and the lessons we have learned as a result.

2.4 Related Systems

An important question to ask about PlanetLab is whether its specific design requirements make it unique, or if our experiences can apply to other systems. Our response is that PlanetLab shares "points of pain" with three similar systems—ISPs, hosting centers, and the GRID—but pushes the envelope relative to each.

First, PlanetLab is like an ISP in that it has many points-of-presence and carries traffic to from the rest of the Internet. Like ISPs (but unlike hosting centers and the GRID), PlanetLab has to provide mechanisms that can by used to identify and stop disruptive traffic. PlanetLab goes beyond traditional ISPs, however, in that it has to deal with arbitrary (and experimental) network services, not just packet forwarding.

Second, PlanetLab is like a hosting center in that its nodes support multiple VMs, each on behalf of a different user. Like a hosting center (but unlike the GRID or ISPs), PlanetLab has to provide mechanisms that enforce isolation between VMs. PlanetLab goes beyond hosting centers, however, because it includes third-party services that manage other VMs, and because it must scale to large numbers of VMs with limited resources.

Third, PlanetLab is like the GRID in that its resources are owned by multiple autonomous organizations. Like the GRID (but unlike an ISP or hosting center), PlanetLab has to provide mechanisms that allow one organization to grant users at another organization the right to use its resources. PlanetLab goes far beyond the GRID, however, in that it scales to hundreds of "peering" organizations by avoiding pair-wise agreements.

PlanetLab faces new and unique problems because it is at the intersection of these three domains. For example, combining multiple independent VMs with a single IP address (hosting center) and the need to trace disruptive traffic back to the originating user (ISP) results in a challenging problem. PlanetLab's experiences will be valuable to other systems that may emerge where any of these domains intersect, and may in time influence the direction of hosting centers, ISPs, and the GRID as well.

3 Slice Management

This section describes the slice management architecture that evolved over the past four years. While the discussion includes some details, it primarily focuses on the design decisions and the factors that influenced them.

3.1 Trust Assumptions

Given that PlanetLab sites and users span multiple organizations (R3), the first design issue was to define the underlying trust model. Addressing this issue required that we identify the key principals, explicitly state the trust assumptions among them, and provide mechanisms that are consistent with this trust model.

Over 300 autonomous organizations have contributed nodes to PlanetLab (they each require control over the nodes they own) and over 300 research groups want to deploy their services across PlanetLab (the node owners need assurances that these services will not be disruptive). Clearly, establishing 300×300 pairwise trust relationships is an unmanageable task, but it is well-understood that a trusted intermediary is an effective way to manage such an N×N problem.

PLC is one such trusted intermediary: node owners trust PLC to manage the behavior of VMs that run on their nodes while preserving their autonomy, and researchers trust PLC to provide access to a set of nodes that are capable of hosting their services. Recognizing this role for PLC, and organizing the architecture around

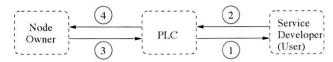


Figure 1: Trust relationships among principals.

it, is the single most important aspect of the design beyond the simple model presented in Section 2.2.

With this backdrop, the PlanetLab architecture recognizes three main principals:

- PLC is a trusted intermediary that manages nodes on behalf a set of owners, and creates slices on those nodes on behalf of a set of users.
- An owner is an organization that hosts (owns) PlanetLab nodes. Each owner retains ultimate control over their own nodes, but delegates management of those nodes to the trusted PLC intermediary. PLC provides mechanisms that allow owners to define resource allocation policies on their nodes.
- A user is a researcher that deploys a service on a set
 of PlanetLab nodes. PlanetLab users are currently
 individuals at research organizations (e.g., universities, non-profits, and corporate research labs), but
 this is not an architectural requirement. Users create
 slices on PlanetLab nodes via mechanisms provided
 by the trusted PLC intermediary.

Figure 1 illustrates the trust relationships between node owners, users, and the PLC intermediary. In this figure:

- PLC expresses trust in a user by issuing it credentials that let it access slices. This means that the user must adequately convince PLC of its identity (e.g., affiliation with some organization or group).
- A user trusts PLC to act as its agent, creating slices on its behalf and checking credentials so that only that user can install and modify the software running in its slice.
- An owner trusts PLC to install software that is able to map network activity to the responsible slice. This software must also isolate resource usage of slices and bound limit slice behavior.
- 4. PLC trusts owners to keep their nodes physically secure. It is in the best interest of owners to not circumvent PLC (upon which it depends for accurate policing of its nodes). PLC must also verify that every node it manages actually belongs to an owner with which it has an agreement.

Given this model, the security architecture includes the following mechanisms. First, each node boots from an immutable file system, loading (1) a boot manager program, (2) a public key for PLC, and (3) a node-specific secret key. We assume that the node is physically secured by the owner in order to keep the key secret, although a hardware mechanism such as TCPA could also be leveraged. The node then contacts a boot server running at PLC, authenticates the server using the public key, and uses HMAC and the secret key to authenticate itself to PLC. Once authenticated, the boot server ensures that the appropriate VMM and the NM are installed on the node, thus satisfying the fourth trust relationship.

Second, once PLC has vetted an organization through an off-line process, users at the site are allowed to create accounts and upload their private keys. PLC then installs these keys in any VMs (slices) created on behalf of those users, and permits access to those VMs via ssh. Currently, PLC requires that new user accounts are authorized by a *principal investigator* associated with each site—this provides some degree of assurance that accounts are only created by legitimate users with a connection to a particular site, thus satisfying the first trust relationship.

Third, PLC runs an auditing service that records information about all packet flows coming out of the node. The auditing service offers a public, web-based interface on each node, through which anyone that has received unwanted network traffic from the node can determine the responsible users. PLC archives this auditing information by periodically downloading the audit log.

3.2 Virtual Machines and Resource Pools

Given the requirement that PlanetLab support long-lived slices (R1) and accommodate scarce resources (R5), the second design decision was to decouple slice creation from resource allocation. In contrast to a hosting center that might create a VM and assign it a fixed set of resources as part of an SLA, PlanetLab creates new VMs without regard for available resources—each such VM is given a fair share of the available resources on that node whenever it runs—and then expects slices to engage one or more brokerage services to acquire resources.

To this end, the NM supports two abstract objects: virtual machines and resource pools. The former is a container that provides a point-of-presence on a node for a slice. The latter is a collection of physical and logical resources that can be bound to a VM. The NM supports operations to create both objects, and to bind a pool to a VM for some fixed period of time. Both types of objects are specified by a resource specification (rspec), which is a list of attributes that describe the object. A VM can run as soon as it is created, and by default is given a fair share of the node's unreserved capacity. When a resource pool

is bound to a VM, that VM is allocated the corresponding resources for the duration of the binding.

Global management services use these per-node operations to create PlanetLab-wide slices and assign resources to them. Two such service types exist today: *slice creation services* and *brokerage services*. These services can be separate or combined into a single service that both creates and provisions slices. At the same time, different implementations of brokerage services are possible (e.g., market-based services that provide mechanisms for buying and selling resources [10, 14], and batch scheduling services that simply enforce admission control for use of a finite resource pool [7]).

As part of the resource allocation architecture, it was also necessary to define a policy that governs how resources are allocated. On this point, owner autonomy (R4) comes into play: only owners are allowed to invoke the "create resource pool" operation on the NM that runs on their nodes. This effectively defines the one or more "root" pools, which can subsequently be split into subpools and reassigned. An owner can also directly allocate a certain fraction of its node's resources to the VM of a specific slice, thereby explicitly supporting any services the owner wishes to host.

3.3 Delegation

PlanetLab's management architecture was expected to evolve through the introduction of third-party services (R2). We viewed the NM interface as the key feature, since it would support the many third-party creation and brokerage services that would emerge. We regarded PLC as merely a "bootstrap" mechanism that could be used to deploy such new global management services, and thus, we expected PLC to play a reduced role over time.

However, experience showed this approach to be flawed. This is for two reasons, one fundamental and one pragmatic. First, it failed to account for PLC's central role in the trust model of Section 3.1. Maintaining trust relationships among participants is a critical role played by PLC, and one not easily passed along to other services. Second, researchers building new management services on PlanetLab were not interested in replicating all of PLCs functionality. Instead of using PLC to bootstrap a comprehensive suite of management services, researchers wanted to leverage some aspects of PLC and replace others.

To accommodate this situation, PLC is today structured as follows. First, each owner implicitly assigns all of its resources to PLC for redistribution. The owner can override this allocation by granting a set of resources to a specific slice, or divide resources among multiple brokerage services, but by default all resources are allocated to PLC.

Second, PLC runs a slice creation service-called

pl_conf—on each node. This service runs in a standard VM and invokes the NM interface without any additional privilege. It also exports an XML-RPC interface by which anyone can invoke its services. This is important because it means other brokerage and slice creation services can use pl_conf as their point-of-presence on each node rather than have to first deploy their own slice. Originally, the PLC/pl_conf interface was private as we expected management services to interact directly with the node manager. However, making this a well-defined, public interface has been a key to supporting delegation.

Third, PLC provides a front-end—available either as a GUI or as a programmatic interface at www.planet-lab.org—through which users create slices. The PLC front-end interacts with pl_conf on each node with the same XML-RPC interface that other services use.

Finally, PLC supports two methods by which slices are actually instantiated on a set of nodes: direct and delegated. Using the direct method, the PLC front-end contacts pl_conf on each node to create the corresponding VM and assign resources to it. Using delegation, a slice creation service running on behalf of a user contacts PLC for a ticket that encapsulates the right to create a VM or redistribute a pool of resources. A ticket is a signed rspec; in this case, it is signed by PLC. The agent then contacts pl_conf on each node to redeem this ticket, at which time pl_conf validates it and calls the NM to create a VM or bind a pool of resources to an existing VM. The mechanisms just described currently support two slice creation services (PLC and Emulab [34], the latter uses tickets granted by the former), and two brokerage services (Sirius [7] and Bellagio [2], the first of which is granted capacity as part of a root resource allocation decision).

Note that the delegated method of slice creation is push-based, while the direct method is pull-based. With delegation, a slice creation service contacts PLC to retrieve a ticket granting it the right to create a slice, and then performs an XML-RPC call to pl_conf on each node. For a slice spanning a significant fraction of PlanetLab's nodes, an implementation would likely launch multiple such calls in parallel. In contrast, PLC uses a polling approach: each pl_conf contacts PLC periodically to retrieve a set of tickets for the slices it should run.

While the push-based approach can create a slice in less time, the advantage of pull-based approach is that it enables slices to persist across node reinstalls. Nodes cannot be trusted to have persistent state since they are completely reinstalled from time to time due to unrecoverable errors such as corrupt local file systems. The pull-based strategy views all nodes as maintaining only soft state, and gets the definitive list of slices for that node from PLC. Therefore, if a node is reinstalled, all of its slices are automatically recreated. Delegation makes it

possible for others to develop alternative slice creation semantics—for example, a "best effort" system that ignores such problems—but PLC takes the conservative approach because it is used to create slices for essential management services.

3.4 Federation

Given our desire to minimize the centralized elements of PlanetLab (R4), our next design decision was to make it possible for multiple independent PlanetLab-like systems to co-exist and federate with each other. Note that this issue is distinct from delegation, which allows multiple management services to co-exist withing a single Planet-Lab

There are three keys to enabling federation. First, there must be well-defined interfaces by which independent instances of PLC invoke operations on each other. To this end, we observe that our implementation of PLC naturally divides into two halves: one that creates slices on behalf of users and one that manages nodes on behalf of owners, and we say that PLC embodies a *slice authority* and a *management authority*, respectively. Corresponding to these two roles, PLC supports two distinct interfaces: one that is used to create and control slices, and one that is used to boot and manage nodes. We claim that these interfaces are minimal, and hence, define the "narrow waist" of the PlanetLab hourglass.

Second, supporting multiple independent PLCs implies the need to name each instance. It is PLC in its slice authority role that names slices, and its name space must be extended to also name slice authorities. For example, the slice cornell.cobweb is implicitly plc.cornell.cobweb, where plc is the top-level slice authority that approved the slice. (As we generalize the slice name space, we adopt "." instead of "_" as the delimiter.) Note that this model enables a hierarchy of slice authorities, which is in fact already the case with plc.cornell, since PLC trusts Cornell to approve local slices (and the users bound to them).

This generalization of the slice naming scheme leads to several possibilities:

- PLC delegates the ability to create slices to regional slice authorities (e.g., plc.japan.utokyo.ubiq);
- organizations create "private" PlanetLab's (e.g., epfl.chawla) that possibly peer with each other, or with the "public" PlanetLab; and
- alternative "root" naming authorities come into existence, such as one that is responsible for commercial (for-profit) slices (e.g., com.startup.voip).

The third of these is speculative, but the first two scenarios have already happened or are in progress, with five private PlanetLabs running today and two regional slice

Service	Lines of Code	Language
Node Manager	2027	Python
Proper	5752	С
pl_conf	1975	Python
Sirius	850	Python
Stork	12803	Python
CoStat + CoMon	1155	С
PlanetFlow	5932	С

Table 1: Source lines of code for various management services

authorities planned for the near future. Note that there must be a single global naming authority that ensures all top-level slice authority names are unique. Today, PLC plays that role.

The third key to federation is to design pl_conf so that it is able to create slices on behalf of many different slice authorities. Node owners allocate resources to the slice authorities they want to support, and configure pl_conf to accept tickets signed by slice authorities that they trust. Note that being part of the "public" PlanetLab carries the stipulation that a certain minimal amount of capacity be set aside for slices created by the PLC slice authority, but owners can reserve additional capacity for other slice authorities and for individual slices.

3.5 Least Privilege

We conclude our description of PlanetLab's management architecture by focusing on the node-centric issue of how management functionality has been factored into self-contained services, moved out of the NM and isolated in their own VMs, and granted minimal privileges.

When PlanetLab was first deployed, all management services requiring special privilege ran in a single root VM as part of a monolithic node manager. Over time, stand-alone services have been carved off of the NM and placed in their own VMs, multiple versions of some services have come and gone, and new services have emerged. Today, there are five broad classes of management services. The following summarizes one particular "suite" of services that a user might engage; we also identify alternative services that are available.

Slice Creation Service: pl_conf is the default slice creation service. It requires no special privilege: the node owner creates a resource pool and assigns it to pl_conf when the node boots. Emulab [34] offers an alternative slice creation service that uses tickets granted by PLC and redeemed by pl_conf.

Brokerage Service: Sirius [7] is the most widely used brokerage service. It performs admission control on a resource pool set aside for one-hour experiments.

Sirius requires no special privilege: pl_conf allocates a sub-pool of resources to Sirius. Bellagio [2] and Tycoon [14] are alternative market-based brokerage services that are initialized in the same way.

Monitoring Service: CoStat is a low-level instrumentation program that gathers data about the state of the local node. It is granted the ability to read /proc files that report data about the underlying VMM, as well as the right to execute scripts (e.g., ps and top) in the root context. Multiple additional services—e.g., CoMon [31], PsEPR [5], SWORD [18]—then collect and process this information on behalf of users. These services require no additional privilege.

Environment Service: Stork [12] deploys, updates, and configures services and experiments. Stork is granted the right to mount the file system of a client slice, which Stork then uses to install software packages required by the slice. It is also granted the right to mark a file as immutable, so that it can safely be shared among slices without any slice being able to modify the file. Emulab and AppManager [28] provide alternative environment services without extra privilege; they simply provide tools for uploading software packages.

Auditing Service: PlanetFlow [11] is an auditing service that logs information about packet flows, and is able to map externally visible network activity to the responsible slice. PlanetFlow is granted the right to run ulogd in the root context to retrieve log information from the VMM.

The need to grant narrowly-defined privileges to certain management services has led us to define a mechanism called Proper (PRivileged OPERation) [17]. Proper uses an ACL to specify the particular operations that can be performed by a VM that hosts a management service, possibly including argument constraints for each operation. For example, the CoStat monitoring service gathers various statistics by reading /proc files in the root context, so Proper constrains the set of files that can be opened by CoStat to only the necessary directories. For operations that affect other slices directly, such as mounting the slice's file system or executing a process in that slice, Proper also allows the target slice to place additional constraints on the operations that can be performed e.g., only a particular directory may be mounted by Stork. In this way we are able to operate each management service with a small set of additional privileges above a normal slice, rather than giving out coarse-grained capabilities such as those provided by the standard Linux kernel, or co-locating the service in the root context.

Finally, Table 1 quantifies the impact of moving functionality out of the NM in terms of lines-of-code. The LOC data is generated using David A. Wheeler's 'SLOC-Count'. Note that we show separate data for Proper and the rest of the node manager; Proper's size is in part a function of its implementation in C.

One could argue that these numbers are conservative, as there are additional services that this list of management services employ. For example, CoBlitz is a large file transfer mechanism that is used by Stork and Emulab to disseminate large files across a set of nodes. Similarly, a number of these services provide a web interface that must run on each node, which would greatly increase the size of the TCB if the web server itself had to be included in the root context.

4 Resource Allocation

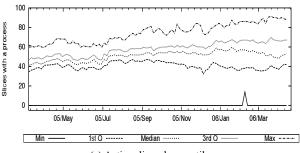
One of the most significant challenges for PlanetLab has been to maximize the platform's utility for a large user community while dealing with the reality of limited resources. This challenge has led us to a model of weak resource isolation between slices. We implement this model through fair sharing of CPU and network bandwidth, simple mechanisms to avoid the worst kinds of interference on other resources like memory and disk, and tools to give users information about resource availability on specific nodes. This section reports our experiences with this model in practice, and describes some of the techniques we've adopted to make the system as effective and stable as possible.

4.1 Workload

PlanetLab supports a workload mixing one-off experiments with long-running services. A complete characterization of this workload is beyond the scope of this paper, but we highlight some important aspects below.

CoMon—one of the performance-monitoring services running on PlanetLab—classifies a slice as *active* on a node if it contains a process, and *live* if, in the last five minutes, it used at least 0.1% (300ms) of the CPU. Figure 2 shows, by quartile, the number of active and live slices across PlanetLab during the past year. Each graph shows five lines; 25% of PlanetLab nodes have values that fall between the first and second lines, 25% between the second and third, and so on.

Looking at each graph in more detail, Figure 2(a) illustrates that the number of active slices on most PlanetLab nodes has grown steadily. The median active slice count has increased from 40 slices in March 2005 to the mid-50s in April 2006, and the maximum number of active slices has increased from 60 to 90. PlanetLab nodes can support large numbers of mostly idle slices because each VM is very lightweight. Additionally, the data shows that 75% of PlanetLab nodes have consistently had at least 40



(a) Active slices, by quartile

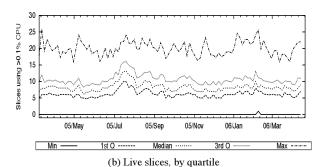


Figure 2: Active and live slices on PlanetLab

active slices during the past year.

Figure 2(b) shows the distribution of live slices. Note that at least 50% of PlanetLab nodes consistently have a live slice count within two of the median. Additional data indicates that this is a result of three factors. First, some monitoring slices (like CoMon and PlanetFlow) are live everywhere, and so create a lower bound on the number of live slices. Second, most researchers do not appear to greedily use more nodes than they need; for example, only 10% of slices are deployed on all nodes, and 60% are deployed on less than 50 nodes. We presume researchers are self-organizing their services and experiments onto disjoint sets of nodes so as to distribute load, although there are a small number of popular nodes that support over 25 live slices. Third, the slices that are deployed on all nodes are not live on all of them at once. For instance, in April 2006 we observed that CoDeeN was active on 436 nodes but live on only 269. Robust (and adaptive) long-running services are architected to dynamically balance load to less utilized nodes [33, 26].

Of course we did not know what PlanetLab's work-load would look like when we made many early design decisions. As reported in Section 2.2, one such decision was to use Linux+Vservers as the VMM, primarily because of the maturity of the technology. Since this time, alternatives like Xen have advanced considerably, but we have not felt compelled to reconsider this decision. A key reason is that PlanetLab nodes run up to 25 live VMs,

and up to 90 active VMs, at at time. This is possible because we could build a system that supports resource overbooking and graceful degradation on a framework of Vserver-based VMs. In contrast, Xen allocates specific amounts of resources, such as physical memory and disk, to each VM. For example, on a typical PlanetLab node with 1GB memory, Xen can support only 10 VMs with 100MB memory each, or 16 with 64MB memory. Therefore, it's not clear how a PlanetLab based on Xen could support our current user base. Note that the management architecture presented in the previous section is general enough to support multiple VM types (and a Xen prototype is running in the lab), but resource constraints make it likely that most PlanetLab slices will continue to use Vservers for the foreseeable future.

4.2 Graceful Degradation

PlanetLab's usage model is to allow as many users on a node as want to use it, enable resource brokers that are able to secure guaranteed resources, and gracefully degrade the node's performance as resources become over utilized. This section describes the mechanisms that support such behavior and evaluates how well they work.

4.2.1 CPU

The earliest version of PlanetLab used the standard Linux CPU scheduler, which provided no CPU isolation between slices: a slice with 400 Java threads would get 400 times the CPU of a slice with one thread. This situation occasionally led to collapse of the system and revealed the need for a slice-aware CPU scheduler.

Fair share scheduling [32] does not collapse under load, but rather supports graceful degradation by giving each scheduling container proportionally fewer cycles. Since mid-2004, PlanetLab's CPU scheduler has performed fair sharing among slices. During that time, however, PlanetLab has run three distinct CPU schedulers: v2 used the SILK scheduler [3], v3.0 introduced CKRM (a community project in its early stages), and v3.2 (the current version) uses a modification of Vserver's CPU rate limiter to implement fair sharing and reservations. The question arises, why so many CPU schedulers?

The answer is that, for the most part, we switched CPU schedulers for reasons other than scheduling behavior. We switched from SILK to CKRM to leverage a community effort and reduce our code maintenance burden. However, at the time we adopted it, CKRM was far from production quality and the stability of PlanetLab suffered as a result. We then dropped CKRM and wrote another CPU scheduler, this time based on small modifications to the Vservers code that we had already incorporated into the PlanetLab kernel. This CPU scheduler gave us the capability to provide slices with CPU reservations as well as shares, which we lacked with SILK and CKRM. Per-

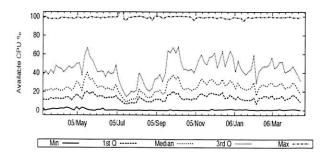


Figure 3: CPU % Available on PlanetLab

haps more importantly, the scheduler was more robust, so PlanetLab's stability dramatically improved, as shown in Section 5. We are solving the code maintenance problem by working with the Vservers developers to incorporate our modifications into their main distribution.

The current (v3.2) CPU scheduler implements fair sharing and work-conserving CPU reservations by overlaying a token bucket filter on top of the standard Linux CPU scheduler. Each slice has a token bucket that accumulates tokens at a specified rate; every millisecond, the slice that owns the running process is charged one token. A slice that runs out of tokens has its processes removed from the runqueue until its bucket accumulates a minimum amount of tokens. This filter was already present in Vservers, which used it to put an upper bound on the amount of CPU that any one VM could receive; we simply modified it to provide a richer set of behaviors.

The rate that tokens accumulate depends on whether the slice has a reservation or a share. A slice with a reservation accumulates tokens at its reserved rate: for example, a slice with a 10% reservation gets 100 tokens per second, since a token entitles it to run a process for one millisecond. The default share is actually a small reservation, providing the slice with 32 tokens every second, or 3% of the total capacity.

The main difference between reservations and shares occurs when there are runnable processes but no slice has enough tokens to run: in this case, slices with shares are given priority over slices with reservations. First, if there is a runnable slice with shares, tokens are given out fairly to all slices with shares (i.e., in proportion to the number of shares each slice has) until one can run. If there are no runnable slices with shares, then tokens are given out fairly to slices with reservations. The end result is that the CPU capacity is effectively partitioned between the two classes of slices: slices with reservations get what they've reserved, and slices with shares split the unreserved capacity of the machine proportionally.

CoMon indicates that the average PlanetLab node has its CPU usage pegged at 100% all the time. However, fair sharing means that an individual slice can still ob-

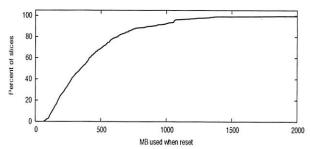


Figure 4: CDF of memory consumed when slice reset

tain a significant percentage of the CPU. Figure 3 shows, by quartile, the CPU availability across PlanetLab, obtained by periodically running a spinloop in the CoMon slice and observing how much CPU it receives. The data shows large amounts of CPU available on PlanetLab: at least 10% of the CPU is available on 75% of nodes, at least 20% CPU on 50% of nodes, and at least 40% CPU on 25% of nodes.

4.2.2 Memory

Memory is a particularly scarce resource on PlanetLab, and we were faced with with chosing between four designs. One is the default Linux behavior, which either kernel panics or randomly kills a process when memory becomes scarce. This clearly does not result in graceful degradation. A second is to statically allocate a fixed amount of memory to each slice. Given that there are up to 90 active VMs on a node, this would imply an impractically small 10MB allocation for each VM on the typical node with 1GB of memory. A third option is to explicitly allocate memory to live VMs, and reclaim memory from inactive VMs. This implies the need for a control mechanism, but globally synchronizing such a mechanism across PlanetLab (i.e., to suspend a slice) is problematic at fine-grained time scales. The fourth option is to dynamically allocate memory to VMs on demand, and react in a more predictable way when memory is scarce.

We elected the fourth option, implementing a simple watchdog daemon, called pl_mom, that resets the slice consuming the most physical memory when swap has almost filled. This penalizes the memory hog while keeping the system running for everyone else.

Although pl_mom was noticably active when first deployed—as users learned to not keep log files in memory and to avoid default heap sizes—it now typically resets an average of 3-4 VMs per day, with higher rates during heavy usage (e.g., major conference deadlines). For example, 200 VMs were reset during the two week runup to the OSDI deadline. We note, however, that roughly one-third of these resets were on under-provisioned nodes (i.e., nodes with less than 1GB of memory).

Figure 4 shows the cumulative distribution function of

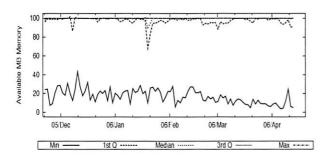


Figure 5: Memory availability on PlanetLab

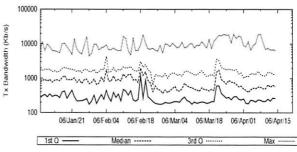
how much physical memory individual VMs were consuming when they were reset between November 2004 and April 2006. We note that about 10% of the resets (corresponding largely to the first 10% of the distribution) occurred on nodes with less than 1GB memory, where memory pressure was tighter. Over 80% of all resets had been allocated at least 128MB. Half of all resets occurred when the slice was using more than 400MB of memory, which on a shared platform like PlanetLab indicates either a memory leak or poor experiment design (e.g., a large in-memory logfile).

Figure 5 shows CoMon's estimate of how many MB of memory are available on each PlanetLab node. CoMon estimates available memory by allocating 100MB, touching random pages periodically, and then observing the size of the in-memory working set over time. This serves as a gauge of memory pressure, since if physical memory is exhausted and another slice allocates memory, these pages would be swapped out. The CoMon data shows that a slice can keep a 100MB working set in memory on at least 75% of the nodes (since only the minimum and first quartile line are really visible), so it appears that there is not as much memory pressure on PlanetLab as we expected. This also reinforces our intuition that pl_mom resets slices mainly on nodes with too little memory or when the slice's application has a memory leak.

4.2.3 Bandwidth

Hosting sites can cap the maximum rate at which the local PlanetLab nodes can send data. PlanetLab fairly shares the bandwidth under the cap among slices, using Linux's Hierarchical Token Bucket traffic filter [15]. The node bandwidth cap allows sites to limit the *peak* rate at which nodes send data so that PlanetLab slices cannot completely saturate the site's outgoing links.

The *sustained* rate of each slice is limited by the pl_mom watchdog daemon. The daemon allows each slice to send a quota of bytes each day at the node's cap rate, and if the slice exceeds its quota, it imposes a much smaller cap for the rest of the day. For example, if the slice's quota is 16GB/day, then this corresponds to a sustained rate of 1.5Mbps; once the slice sends more than



(a) Transmit bandwidth in Kb s, by quartile

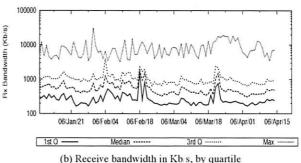


Figure 6: Sustained network rates on PlanetLab

16GB, it is capped at 1.5Mbps until midnight GMT. The goal is to allow most slices to burst data at the node's cap rate, but prevents slices that are sending large amounts of data from badly abusing the site's local resources.

There are two weaknesses of PlanetLab's bandwidth capping approach. First, some sites pay for bandwidth based on the total amount of traffic they generate per month, and so they need to control the node's sustained bandwidth rather than the peak. As mentioned, pl_mom limits sustained bandwidth, but it operates on a per-slice (rather than per-node) basis and cannot currently be controlled by the sites. Second, PlanetLab does not currently cap *incoming* bandwidth. Therefore, PlanetLab nodes can still saturate a bottleneck link by downloading large amounts of data. We are currently investigating ways to fix both of these limitations.

Figure 6 shows, by quartile, the sustained rates at which traffic is sent and received on PlanetLab nodes since January 2006. These are calculated as the sums of the average transmit and receive rates for all the slices of the machine over the last 15 minutes. Note that the y axis is logarithmic, and the Minimum line is omitted from the graph. The typical PlanetLab node transmits about 1Mb/s and receives 500Kb/s, corresponding to about 10.8GB/day sent and 5.4GB/day received. These numbers are well below the typical node bandwidth cap of 10Mb/s. On the other hand, some PlanetLab nodes do actually have sustained rates of 10Mb/s both ways.

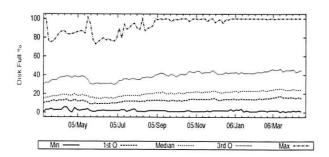


Figure 7: Disk usage, by quartile, on PlanetLab

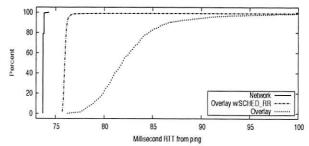


Figure 8: RTT CDF on network, overlay with and without SCHED_RR

4.2.4 Disk

PlanetLab nodes do not provide permanent storage: data is not backed up, and any node may be reinstalled without warning. Services adapt to this environment by treating disk storage as a cache and storing permanent data elsewhere, or else replicating data on multiple nodes. Still, a PlanetLab node that runs out of disk space is essentially useless. In our experience, disk space is usually exhausted by runaway log files written by poorly-designed experiments. This problem was mitigated, but not entirely solved, by the introduction of per-slice disk quotas in June 2005. The default quota is 5GB, with larger quotas granted on a case-by-case basis.

Figure 7 shows, by quartile, the disk utilization on PlanetLab. The visible dip shortly after May 2005 is when quotas were introduced. We note that, though disk utilization grows steadily over time, 75% of Planetlab nodes still have at least 50% of the disks free. Some PlanetLab nodes do occasionally experience full disks, but most are old nodes that do not meet the current system requirements.

4.2.5 Jitter

CPU scheduling latency can be a serious problem for some PlanetLab applications. For example, in a packet forwarding overlay, the time between when a packet arrives and when the packet forwarding process runs will appear as added network latency to the overlay clients. Likewise, many network measurement applications assume low scheduling latency in order to produce precisely spaced packet trains. Many measurement applications can cope with latency by knowing which samples to trust and which must be discarded, as described in [29]. Scheduling latency is more problematic for routing overlays, which may have to drop incoming packets.

A simple experiment indicates how scheduling latency can affect applications on PlanetLab. We deploy a packet forwarding overlay, constructed using the Click modular software router [13], on six Planetlab nodes co-located at Abilene PoPs between Washington, D.C. and Seattle. Our experiment then uses ping packets to compare the

RTT between the Seattle and D.C. nodes on the network and on the six-hop overlay. Each of the six PlanetLab nodes running our overlay had load averages between 2 and 5, and between 5 and 8 live slices, during the experiment. We observe that the network RTT between the two nodes is a constant 74ms over 1000 pings, while the overlay RTT varies between 76ms and 135ms. Figure 8 shows the CDF of RTTs for the network (leftmost curve) and the overlay (rightmost curve). The overlay CDF has a long tail that is chopped off at 100ms in the graph.

There are several reasons why the overlay could have its CPU scheduling latency increased, including: (1) if another task is running when a packet arrives, Click must wait to forward the packet until the running task blocks or exhausts its timeslice; (2) if Click is trying to use more than its "fair share", or exceeds its CPU guarantee, then its token bucket CPU filter will run out of tokens and it will be removed from the runqueue until it acquires enough tokens to run; (3) even though the Click process may be runnable, the Linux CPU scheduler may still decide to schedule a different process; and (4) interrupts and other kernel activity may preempt the Click process or otherwise prevent it from running.

We can attack the first three sources of latency using existing scheduling mechanisms in PlanetLab. First, we give the overlay slice a CPU reservation to ensure that it will never run out of tokens during our experiment. Second, we use chrt to run the Click process on each machine with the SCHED_RR scheduling policy, so that it will immediately jump to the head of the runqueue and preempt any running task. The Proper service described in Section 3.5 enables our slice to run the privileged chrt command on each PlanetLab node.

The middle curve in Figure 8 shows the results of rerunning our experiment with these new CPU scheduling parameters. The overhead of the Click overlay, around 3ms, is clearly visible as the difference between the two left-most curves. In the new experiment, about 98% of overlay RTTs are within 3ms of the underlying network RTT, and 99% are within 6ms. These CPU scheduling mechanisms are employed by *PL-VINI*, the VINI (VIr-

tual Network Infrastructure) prototype implemented on PlanetLab, to reduce latency in an overlay network as an artifact of CPU scheduling delay [4].

We note two things. First, the obstacle to making this solution available on PlanetLab is primarily one of policy-choosing which slices should get CPU reservations and bumps to the head of the runqueue, since it is not possible to reduce everyone's latency on a heavily loaded system. We plan to offer this option to shortterm experiments via the Sirius brokerage service, but long-running routing overlays will need to be handled on a case-by-case basis. Second, while our approach can provide low latency to the Click forwarder in our experiment 99% of the time, it does not completely solve the latency problem. We hypothesize that the remaining CPU scheduling jitter is due to the fourth source of latency identified earlier, i.e., kernel activity. If so, we may be able to further reduce it by enabling kernel preemption, a feature already available in the Linux 2.6 kernel.

4.2.6 Remarks

Note that only limited conclusions can be drawn from the fact that there is unused capacity available on PlanetLab nodes. Users are adapting to the behavior of the system (including electing to not use it) and they are writing services that adapt to the available resources. It is impossible to know how many resources would have been used, even by the same workload, had more been available. However, the data does document that PlanetLab's fair share approach is behaving as expected.

5 Operational Stability

The need to maintain a stable system, while at the same time evolving it based on user experience, has been a major complication in designing PlanetLab. This section outlines the general strategies we adopted, and presents data that documents our successes and failures.

5.1 Strategies

There is no easy way to continually evolve a system that is experiencing heavy use. Upgrades are potentially disruptive for at least two reasons: (1) new features introduce new bugs, and (2) interface changes force users to upgrade their applications. To deal with this situation, we adopted three general strategies.

First, we kept PlanetLab's control plane (i.e., the services outlined in Section 3) orthogonal from the OS. This meant that nearly all of the interface changes to the system affected only those slices running management services; the vast majority of users were able to program to a relatively stable Linux API. In retrospect this is an obvious design principle, but when the project began, we believed our main task was to define a new OS interface tailored for wide-area services. In fact, the one example where we deviated from this principle—by changing the

socket API to support *safe raw sockets* [3]—proved to be an operational diaster because the PlanetLab OS looked enough like Linux that any small deviation caused disproportionate confusion.

Second, we leveraged existing software whereever possible. This was for three reasons: (1) to improve the stability of the system; (2) to lower the barrier-to-entry for the user community; and (3) to reduce the amount of new code we had to implement and maintain. This last point cannot be stressed enough. Even modest changes to existing software packages have to be tracked as those packages are updated over time. In our eagerness to reuse rather than invent, we made some mistakes, the most notable of which is documented in the next subsection.

Third, we adopted a well-established practice of rolling new releases out incrementally. This is for the obvious reason—to build confidence that the new release actually worked under realistic loads before updating all nodes—but also for a reason that's somewhat unique to Planet-Lab: some long-running services maintain persistent data repositories, but doing so depends on a minimal number of copies being available at any time. Updates that reboot nodes must happen incrementally if long-running storage services are to survive.

Note that while few would argue with these principles—and it is undoubtedly the case that we would have struggled had we not adhered to them—our experience is that many other factors (some unexpected) had a significant impact on the stability of the system. The rest of this section reports on these operational experiences.

5.2 Node Stability

We now chronicle our experience operating and evolving PlanetLab. Figure 9 illustrates the availability of PlanetLab nodes from September 2004 through April 2006, as inferred from CoMon. The bottom line indicates the PlanetLab nodes that have been up continuously for the last 30 days (stable nodes), the middle line is the count of nodes that came online within the last 30 days, and the top line is all registered PlanetLab nodes. Note that the difference between the bottom and middle lines represents the "churn" of PlanetLab over a month's time; and the difference between the middle and top lines indicates the number of nodes that are offline. The vertical lines in Figure 9 are important dates, and the letters at the top of the graph let us refer to the intervals between the dates.

There have clearly been problems providing the community with a stable system. Figure 9 illustrates several reasons for this:

 Sometimes instability stems from the community stressing the system in new ways. In Figure 9, interval A is the run-up to the NSDI'05 deadline. During this time, heavy use combined with memory leaks in

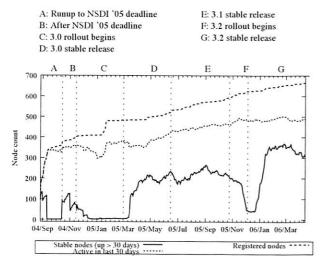


Figure 9: Node Availability

some experiments caused kernel panics due to Outof-Memory errors. This is the common behavior of Linux when the system runs out of memory and swap space. pl_mom (Section 4.2.2) was introduced in response to this experience.

- Software upgrades that require a reboot obviously effect the set of stable nodes (e.g., intervals C and D), but installing buggy software has a longer-term affect on stability. Interval C shows the release of PlanetLab 3.0. Although the release had undergone extensive off-line testing, it had bugs and a relatively long period of instability followed. PlanetLab was still usable during this period but nodes rebooted at least once per month.
- The pl_mom watchdog is not perfect. There is a slight dip in the number of stable core nodes (bottom line) in interval D, when about 50 nodes were rebooted because of a slice with a fast memory leak; as memory pressure was already high on those nodes, pl_mom could not detect and reset the slice before the nodes ran out of memory.

We note, however, that the 3.2 software release from late December 2005 is the best so far in terms of stability: as of February 2006, about two-thirds of active Planet-Lab nodes have been up for at least a month. We attribute most of this to abandoning CKRM in favor of the Vservers native resource management framework and a new CPU scheduler.

One surprising fact to emerge from Figure 9 is that a lot of PlanetLab nodes are dead (denoted by the difference between the top and middle lines). Research organizations gain access to PlanetLab simply by hooking up two machines at their local site. This formula for growth

has worked quite well: a low barrier-to-entry provided the right incentive to grow PlanetLab. However, there have never been well-defined incentives for sites to keep nodes online. Providing such incentives is obviously the right thing to do, but we note that that the majority of the off-line nodes are at sites that no longer have active slices—and at the time of this writing only 12 sites had slices but no functioning nodes—so it's not clear what incentive will work.

Now that we have reached a fairly stable system, it becomes interesting to study the "churn" of nodes that are active yet are not included in the stable core. We find it useful to differentiate between three categories of nodes: those that came up that day (and stayed up), those that went down that day (and stayed down), and those that have rebooted at least once. Our experience is that, on a typical day, about 2 nodes come up, about 2 nodes go down, and about 4 nodes reboot. On 10% of days, at least 6 nodes come up or go down, and at least 8 nodes reboot.

Looking at the archives of the support and planetlab-users mailing lists, we are able to identify the most common reasons nodes come up or go down: (1) a site takes its nodes offline to move them or change their network configuration, (2) a site takes its nodes offline in response to a security incident, (3) a site accidently changes a network configuration that renders its nodes unreachable, or (4) a node goes offline due to a hardware failure. The last is the most common reason for nodes being down for an extended period of time; the third reason is the most frustrating aspect of operating a system that embeds its nodes in over 300 independent IT organizations.

Understanding the relative frequency of different sorts of site events may be important for designers of other large-scale distributed systems; this is a topic for further study.

5.3 Security Complaints

Of the operational issues that PlanetLab faces, responding to security complaints is perhaps the most interesting, if only because of what they say about the current state of the Internet. We comment on three particular types of complaints.

The most common complaints are the result of IDS alerts. One frequent scenario corresponds to a perceived DoS attack. These are sometimes triggered by a poorly designed experiment (in which case the responsible researchers are notified and expected to take corrective action), but they are more likely to be triggered by totally innocent behavior (e.g., 3 unsolicited UDP packets have triggered the threat of legal action). In other cases, the alerts are triggered by simplistic signitures for malware that could not be running on our Linux-based environment. In general, we observe that any traffic that devi-

ates from a rather narrow range of acceptable behavior is increasingly viewed as suspect, which makes innovating with new types of network services a challenge.

An increasingly common type of complaint comes from home users monitoring their firewall logs. They see connections to PlanetLab nodes that they do not recognize, assume PlanetLab has installed spyware on their machines, and demand that it be removed. In reality, they have unknowingly used a service (e.g., a CDN) that has imposed itself between them and a server. Receiving packets from a location service that also probes the client to select the most appropriate PlanetLab node to service a request only exacerbates the situation [35, 9]. The takeaway is that even individual users are becoming increasingly security-senstive (if less security-sophisticated than their professional counterparts), which makes the task of deploying alternative services increasingly problematic.

Finally, PlanetLab nodes are sometimes identified as the source or sink of illegal content. In reality, the content is only cached on the node by a slice running a CDN service, but an overlay node looks like an end node to the rest of the Internet. PlanetLab staff use PlanetFlow to identify the responsible slice, which in turn typically maintains a log that can be used to identify the ultimate source or destination. This information is passed along to the authorities, when appropriate. While many hosting sites are justifiably gun-shy about such complaints, the main lesson we have learned is that trying to police content is not a viable solution. The appropriate approach is to be responsive and cooperative when complaints are raised.

6 Discussion

Perhaps the most fundamental issue in PlanetLab's design is how to manage trust in the face of pressure to decentralize the system, where decentralization is motivated by the desire to (1) give owners autonomous control over their nodes and (2) give third-party service developers the flexibility they need to innovate.

At one end of the spectrum, individual organizations could establish bilateral agreements with those organizations that they trust, and with which they are willing to peer. The problem with such an approach is that reaching the critical mass needed to foster a large-scale deployment has always proved difficult. PlanetLab started at the other end of the spectrum by centralizing trust in a single intermediary—PLC—and it is our contention that doing so was necessary to getting PlanetLab off the ground. To compensate, the plan was to decentralize the system through two other means: (1) users would delegate the right to manage aspects of their slices to third-party services, and (2) owners would make resource allocation decisions for their nodes. This approach has had mixed success, but it is important to ask if these limitations are fun-

damental or simply a matter of execution.

With respect to owner autonomy, all sites are allowed to set bandwidth caps on their nodes, and sites that have contributed more than the two-node minimum required to join PlanetLab are allowed to give excess resources to favored slices, including brokerage services that redistribute those resources to others. In theory, sites are also allowed to blacklist slices they do not want running locally (e.g., because they violate the local AUP), but we have purposely not advertised this capability in an effort to "unionize" the research community: take all of our experiments, even the risky ones, or take none of them. (As a compromise, some slices voluntarily do not run on certain sites so as to not force the issue.) The interface by which owners express their wishes is clunky (and sometimes involves assistance from the PlanetLab staff), but there does not seem to be any architectural reason why this approach cannot provide whatever control over resource allocation that owners require (modulo meeting requirements for joining PlanetLab in the first place).

With respect to third-party management services, success has been much more mixed. There have been some successes—Stork, Sirius, and CoMon being the most notable examples—but this issue is a contentious one in the PlanetLab developer community. There are many possible explanations, including there being few incentives and many costs to providing 24.7 management services; users preferring to roll their own management utilities rather than learn a third-party service that doesn't exactly satisfy their needs; and the API being too much of a moving target to support third-party development efforts.

While these possibilities provide interesting fodder for debate, there is a fundamental issue of whether the centralized trust model impacts the ability to deploy third-party management services. For those services that require privileged access on a node (see Section 3.5) the answer is yes—the PLC support staff must configure Proper to grant the necessary privilege(s). While in practice such privileges have been granted in all cases that have not violated PlanetLab's underlying trust assumptions or jeopardized the stability of the operational system, this is clearly a limitation of the architecture.

Note that choice is not just limited to what management services the central authority approves, but also to what capabilities are included in the core system—e.g., whether each node runs Linux, Windows, or Xen. Clearly, a truly scalable system cannot depend on a single trusted entity making these decisions. This is, in fact, the motivation for evolving PlanetLab to the point that it can support federation. To foster federation we have put together a software distribution, called MyPLC, that allows anyone to create their own private PlanetLab, and potentially federate that PlanetLab with others (including the current "public" PlanetLab).

This returns us to the original issue of centralized versus decentralized trust. The overriding lesson of PlanetLab is that a centralized trust model was essential to achieving some level of critical mass—which in turn allowed us to learn enough about the design space to define a candidate minimal interface for federation—but that it is only by federating autonomous instances that the system will truly scale. Private PlanetLabs will still need bilateral peering agreements with each other, but there will also be the option of individual PlanetLabs scaling internally to non-trivial sizes. In other words, the combination of bilateral agreements and trusted intermediaries allows for flexible aggregation of trust.

7 Conclusions

Building PlanetLab has been a unique experience. Rather than leveraging a new mechanism or algorithm, it has required a synthesis of carefully selected ideas. Rather than being based on a pre-conceived design and validated with controlled experiments, it has been shaped and proven through real-world usage. Rather than be designed to function within a single organization, it is a large-scale distributed system that must be cognizant of its place in a multi-organization world. Finally, rather than having to satisfy only quantifiable technical objectives, its success has depended on providing various communities with the right incentives and being equally responsive to conflicting and difficult-to-measure requirements.

Acknowledgments

Many people have contributed to PlanetLab. Timothy Roscoe, Tom Anderson, and Mic Bowman have provided significant input to the definition of its architecture. Several researchers have also contributed management services, including David Lowenthal, Vivek Pai and KyoungSoo Park, John Hartman and Justin Cappos, and Jay Lepreau and the Emulab team. Finally, the contributions of the PlanetLab staff at Princeton—Aaron Klingaman, Mark Huang, Martin Makowiecki, Reid Moran, Faiyaz Ahmed, Brian Jones, and Scott Karlin—have been immeasurable.

We also thank the anonymous referees, and our shepherd, Jim Waldo, for their comments and help in improving this paper.

This work was funded in part by NSF Grants CNS-0520053, CNS-0454278, and CNS-0335214.

References

- ANNAPUREDDY, S., FREEDMAN, M. J., AND MAZIERES, D. Shark: Scaling File Servers via Cooperative Caching. In *Proc. 2nd NSDI* (Boston, MA, May 2005).
- [2] AUYOUNG, A., CHUN, B., NG, C., PARKES, D., SHNEI-DMAN, J., SNOEREN, A., AND VAHDAT, A. Bellagio: An Economic-Based Resource Allocation System for PlanetLab. http://bellagio.ucsd.edu/about.php.

- [3] BAVIER, A., BOWMAN, M., CULLER, D., CHUN, B., KARLIN, S., MUIR, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating System Support for Planetary-Scale Network Services. In *Proc.* 1st NSDI (San Francisco, CA, Mar 2004).
- [4] BAVIER, A., FEAMSTER, N., HUANG, M., PETERSON, L., AND REX-FORD, J. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proc. SIGCOMM* 2006 (Pisa, Italy, Sep 2006).
- [5] BRIETT, P., KNAUERHASE, R., BOWMAN, M., ADAMS, R., NATARAJ, A., SEDAYAO, J., AND SPINDEL, M. A Shared Global Event Propagation System to Enable Next Generation Distributed Services. In *Proc. 1st* WORLDS (San Francisco, CA, Dec 2004).
- [6] CLARK, D. D. The Design Philosophy of the DARPA Internet Protocols. In Proc. SIGCOMM '88 (Stanford, CA, Aug 1988), pp. 106–114.
- [7] DAVID LOWENTHAL. Sirius: A Calendar Service for PlanetLab. http://snowball.cs.uga.edu/dkl/pslogin.php.
- [8] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIERES, D. Democratizing content publication with Coral. In *Proc. 1st NSDI* (San Francisco, CA, Mar 2004).
- [9] FREEDMAN, M. J., LAKSHMINARAYANAN, K., AND MAZIERES, D. OA-SIS: Anycast for Any Service. In *Proc. 3rd NSDI* (San Jose, CA, May 2006).
- [10] FU, Y., CHASE, J., CHUN, B., SCHWAB, S., AND VAHDAT, A. SHARP: An Architecture for Secure Resource Peering. In *Proc. 19th SOSP* (Lake George, NY, Oct 2003).
- [11] HUANG, M., BAVIER, A., AND PETERSON, L. PlanetFlow: Maintaining Accountability for Network Services. ACM SIGOPS Operating Systems Review 40, 1 (Jan 2006).
- [12] JUSTON CAPPOS AND JOHN HARTMAN. Stork: A Software Packagement Management Service for PlanetLab. http://www.cs.arizona.edu/stork.
- [13] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. ACM Transactions on Computer Systems 18, 3 (Aug 2000), 263–297.
- [14] LAI, K., RASMUSSON, L., ADAR, E., SORKIN, S., ZHANG, L., AND HUBERMAN, B. A. Tycoon: An Implemention of a Distributed Market-Based Resource Allocation System. Tech. Rep. arXiv:es.DC/0412038, HP Labs, Palo Alto, CA, USA, Dec. 2004.
- [15] LINUX ADVANCED ROUTING AND TRAFFIC CONTROL. http://lartc.org/.
- [16] Linux VServers Project. http://linux-vserver.org/.
- [17] MUIR, S., PETERSON, L., FIUCZYNSKI, M., CAPPOS, J., AND HART-MAN, J. Privileged Operations in the PlanetLab Virtualised Environment. SIGOPS Operating Systems Review 40, 1 (2006), 75–88.
- [18] OPPENHEIMER, D., ALBRECH, J., PATTERSON, D., AND VAHDAT, A. Distributed Resource Discovery on PlanetLab with SWORD. In *Proc. 1st WORLDS* (San Francisco, CA, 2004).
- [19] PARK, K., AND PAL, V. S. Scale and Performance in the CoBlitz Large-File Distribution Service. In *Proc. 3rd NSDI* (San Jose, CA, May 2006).
- [20] PARK, K., PAI, V. S., PETERSON, L. L., AND WANG, Z. CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups. In *Proc. 6th OSDI* (San Francisco, CA, Dec 2004), pp. 199–214.
- [21] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In Proc. HotNets-1 (Princeton, NJ, Oct 2002).
- [22] PETERSON, L., BAVIER, A., FIUCZYNSKI, M., MUIR, S., AND ROSCOE, T. PlanetLab Architecture: An Overview. Tech. Rep. PDN-06-031, PlanetLab Consortium, Apr 2006.
- [23] RAMASUBRAMANIAN, V., PETERSON, R., AND SIRER, E. G. Corona: A High Performance Publish-Subscribe System for the World Wide Web. In Proc. 3rd NSDI (San Jose, CA, May 2006).

- [24] RAMASUBRAMANIAN, V., AND SIRER, E. G. Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays. In Proc. 1st NSDI (San Francisco, CA, Mar 2004), pp. 99–112.
- [25] RAMASUBRAMANIAN, V., AND SIRER, E. G. The Design and Implementation of a Next Generation Name Service for the Internet. In *Proc. SIGCOMM 2004* (Portland, OR, Aug 2004), pp. 331–342.
- [26] RHEA, S., GODFREY, B., KARP, B., KUBIATOWICZ, J., RATNASAMY, S., SHENKER, S., STOICA, I., AND YU, H. OPENDHT: A Public DHT Service and its Uses. In *Proc. SIGCOMM 2005* (Philadelphia, PA, Aug 2005), pp. 73–84.
- [27] RITCHIE, D. M., AND THOMPSON, K. The UNIX Time-Sharing System. Communications of the ACM 17, 7 (Jul 1974), 365–375.
- [28] RYAN HUEBSCH. PlanetLab Application Manager. http://appmanager.berkeley.intel-research.net/.
- [29] SPRING, N., BAVIER, A., PETERSON, L., AND PAI, V. S. Using PlanetLab for Network Research: Myths, Realities, and Best Practices. In *Proc. 2nd WORLDS* (San Francisco, CA, Dec 2005).
- [30] SPRING, N., WETHERALL, D., AND ANDERSON, T. Scriptroute: A Public Internet Measurement Facility. In Proc. 4th USITS (Seattle, WA, Mar 2003).
- [31] VIVEK PAI AND KYOUNGSOO PARK. CoMon: A Monitoring Infrastructure for PlanetLab. http://comon.cs.princeton.edu.
- [32] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. 1st OSDI* (Monterey, CA, Nov 1994), pp. 1–11.
- [33] WANG, L., PARK, K., PANG, R., PAI, V. S., AND PETERSON, L. L. Reliability and Security in the CoDeeN Content Distribution Network. In Proc. USENIX '04 (Boston, MA, Jun 2004).
- [34] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An Integrated Experimental Environment for Distributed Systems and Networks. In Proc. 5th OSDI (Boston, MA, Dec 2002), pp. 255–270.
- [35] WONG, B., SLIVKINS, A., AND SIRER, E. G. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In Proc. SIGCOMM 2005 (Philadelphia, PA, Aug 2005).
- [36] ZHANG, M., ZHANG, C., PAI, V. S., PETERSON, L. L., AND WANG, R. Y. PlanetSeer: Internet Path Failure Monitoring and Characterization in Wide-Area Services. In *Proc. 6th OSDI* (San Francisco, CA, Dec 2004), pp. 167–182.

iPlane: An Information Plane for Distributed Services

Harsha V. Madhyastha*

Tomas Isdal*

Michael Piatek*

Colin Dixon*

Thomas Anderson*

Arvind Krishnamurthy*

Arun Venkataramani†

Abstract

In this paper, we present the design, implementation, and evaluation of iPlane, a scalable service providing accurate predictions of Internet path performance for emerging overlay services. Unlike the more common black box latency prediction techniques in use today, iPlane adopts a structural approach and predicts end-to-end path performance by composing the performance of measured segments of Internet paths. For the paths we observed, this method allows us to accurately and efficiently predict latency, bandwidth, capacity and loss rates between arbitrary Internet hosts. We demonstrate the feasibility and utility of the iPlane service by applying it to several representative overlay services in use today: content distribution, swarming peer-to-peer filesharing, and voiceover-IP. In each case, using iPlane's predictions leads to improved overlay performance.

Introduction

The Internet by design is opaque to its applications, providing best effort packet delivery with little or no information about the likely performance or reliability characteristics of different paths. While this is a reasonable design for simple client-server applications, many emerging large-scale distributed services depend on richer information about the state of the network. For example, content distribution networks like Akamai [1], Coral [16], and CoDeeN [52] re-direct each client to the replica providing the best performance for that client. Likewise, voice-over-IP systems such as Skype [45] use relay nodes to bridge hosts behind NAT/firewalls, the selection of which can dramatically affect call quality [39]. Peer-to-peer file distribution, overlay multicast, distributed hash tables, and many other overlay services can benefit from peer selection based on different metrics of network performance such as latency, available bandwidth, and loss rate. Finally, the Internet itself can benefit from more information about itself, e.g., ISPs can monitor the global state of the Internet for reachability and root cause analysis, routing instability, and onset of DDoS attacks.

If Internet performance were easily predictable, its opaqueness might be an acceptable state of affairs. However, Internet behavior is well-known to be fickle, with local hot spots, transient (and partial) disconnectivity. and triangle inequality violations all being quite common [41, 2]. Many large-scale services adapt to this state of affairs by building their own proprietary and application-specific information plane. Not only is this redundant, but it prevents new applications from leveraging information already gathered by other applications. The result is often sub-optimal. For example, most implementations of the file distribution tool Bit-Torrent choose peers at random (or at best using round trip latency estimates); since downloads are bandwidthdependent, this can yield suboptimal download times. By some estimates, BitTorrent accounts for roughly a third of backbone traffic [37], so inefficiency at this scale is a serious concern. Moreover, implementing an information plane is often quite subtle, e.g., large-scale probing of end-hosts can raise intrusion alarms in edge networks as the traffic can resemble a DDoS attack. This is the most common source of complaints on PlanetLab [38].

To address this, several research efforts, such as IDMaps [15], GNP [34], Vivaldi [11], Meridian [54], and PlanetSeer [55] have investigated providing a common measurement infrastructure for distributed applications. These systems provide only a limited subset of the metrics of interest, most commonly latency between a pair of nodes, whereas most applications desire richer information such as loss rate and bandwidth. Second, by treating the Internet as a black box, most of these services abstract away network characteristics and atypical behavior—exactly the information of value for troubleshooting as well as improving performance. For example, the most common latency prediction methods use metric embeddings which are fundamentally incapable of predicting detour paths as such paths violate the triangle inequality [41, 56]. More importantly, being agnostic to network structure, they cannot pinpoint failures, identify causes of poor performance, predict the effect of network topology changes, or assist applications with new functionality such as multipath routing.

In this paper, we move beyond mere latency prediction and develop a service to automatically infer sophis-

^{*}Dept. of Computer Science and Engineering, Univ. of Washington

Dept. of Computer Science, Univ. of Massachusetts Amherst

ticated network behavior. We develop an *Information Plane* (*iPlane*) that continuously performs measurements to generate and maintain an annotated map of the Internet with a rich set of link and router attributes. *iPlane* uses structural information such as the router-level topology and autonomous system (AS) topology to predict paths between arbitrary nodes in the Internet. The path predictions are combined with measured characteristics of path segments to predict end-to-end path properties for a number of metrics such as latency, available bandwidth, and loss rate. *iPlane* can also analyze isolated anomalies or obtain a global view of network behavior by correlating observations from different parts of the Internet.

iPlane is designed as a service that distributed applications can query to obtain information about network conditions. Deploying iPlane as a shared service (as opposed to providing a library) has several benefits. First, a common iPlane can exploit temporal and spatial locality of queries across applications to minimize redundant measurement overhead. Second, iPlane can selectively refresh its knowledge of the IP address space based on real query workloads. More generally, iPlane can assimilate measurements made on behalf of all of its clients as well as incorporate information reported by clients to develop a more comprehensive model of Internet behavior over time. We note that all of these arguments have been recognized before [48, 8, 53, 23], however a convincing validation has remained starkly absent.

Our primary contribution is in demonstrating the feasibility of a useful *iPlane*, e.g., we can infer with high accuracy an annotated map of the Internet every six hours with approximately 100Kbps of measurement traffic per PlanetLab node. In addition, we develop:

- A common structural model to predict path properties.
- A measurement infrastructure that is deployed on every active PlanetLab site and almost a thousand traceroute and Looking Glass server vantage points (with a lower intensity of probing).
- A toolkit for using BitTorrent swarms to measure links.
- Case studies of popular systems such as CDNs, peerto-peer file swarming, and VoIP. We show measurable benefits of using iPlane for each of these applications.

iPlane is a modest step towards the vision of a knowledge plane pioneered by Clark et al. [8]. iPlane supplies information about the network and leaves the task of adapting or repairing to the client. Nevertheless, the collection, analysis, and distribution of Internet-scale measurement information is itself a challenging systems engineering problem and the focus of this paper. The goal of gathering a complete picture of the Internet has been recognized earlier in [48]. Our goal is more modest—to

gather a coarse-grained map of the Internet sufficient to be of utility in improving overlay performance.

2 Design

We start by discussing the requirements of an Information Plane for distributed services before presenting our design that meets these requirements.

- Accuracy: iPlane should accurately estimate a rich set of performance metrics such as latency, loss-rate, capacity, and available bandwidth.
- Wide coverage: iPlane must predict the performance of arbitrary Internet paths. Many currently deployed prediction services, such as RON [2] and S³ [29], limit their focus to intra-overlay paths.
- Scalability: iPlane should not impose an undue communication load on its measurement infrastructure.
- Unobtrusiveness: Active probes of end-hosts must be coordinated and performed in an unobtrusive manner in order to minimize the possibility of raising intrusion detection alarms.

2.1 Overview

iPlane is designed to be deployed as an application-level overlay network with the overlay nodes collectively coordinating the task of generating and maintaining an "atlas" of the Internet. The atlas is both extensive and detailed-it comprises the topology of the Internet core and the core's connectivity to representative targets in the edge networks, complete with a rich set of static attributes (such as link delay and link capacity), and recent observations of dynamic properties (such as routes between network elements, path loss rates, and path congestion). iPlane uses systematic active measurements to determine the attributes of the core routers and the links connecting them. In addition, the system performs opportunistic measurements by monitoring actual data transfers to/from end-hosts participating in BitTorrent swarms, thereby exposing characteristics of the edge of the network that typically cannot be obtained from oneway probing, e.g., capacities of access links.

Since it is impractical to probe every Internet end-host to generate the atlas, we cluster end-hosts on the basis of BGP atoms [4]. We approximate a client's performance by a representative target in the same atom as the client. If the client desires greater prediction accuracy, it can voluntarily perform some probes and contribute the paths that it discovers to *iPlane*; multi-homed clients can benefit from such an operational model. *iPlane* uses its collected repository of observed paths to predict end-to-end paths between any pair of end-hosts. This prediction is made by carefully composing partial segments of known Internet paths so as to exploit the similarity of Internet routes [31], i.e., routes from two nearby sources tend to

Technique	Description	Goal	Section
generate probe targets	Obtain prefixes from Routeview's BGP snapshot and cluster groups of prefixes with similar routes.	coverage, scalability	Section 2.2.1
traceroutes from vantage points	PlanetLab nodes probe all targets, while Traceroute/Looking Glass servers issue probes to a small subset of the targets.	map topology, capture path diversity	Section 2.2.1
cluster network interfaces	Identify network interfaces that are in the same AS and geographically colocated.	build structured topology, scalability	Section 2.2.2
frontier algorithm	Schedule measurements of link attributes to PlanetLab nodes such that each link is probed by the vantage point closest to it.	accuracy, balance load	Section 2.3.1
measure link attributes	PlanetLab nodes measure the loss rate, capacity, and available bandwidth over a subset of paths in the Internet core.	annotate topology	Section 2.3.2
opportunistic measurements	Leverage existing applications to discover the structure and performance of edge networks.	minimize obtrusiveness, access link properties	Section 2.4
route composition	Compose segments of observed or reported paths to predict end- to-end paths between a pair of nodes.	path prediction, performance prediction	Section 2.5

Table 1: A summary of techniques used in iPlane.

converge when heading to the same destination. *iPlane* predicts a path by splicing a short path segment from the source to an intersection point from which a path going to the destination has been observed in the atlas. To determine intersections between paths, we cluster interfaces that are owned by the same AS and reside in the same PoP, and deem two paths to have intersected if they pass through the same cluster.

Once a path is predicted, *iPlane* simply composes the measured properties of the constituent path segments to predict the performance of the composite path. For instance, to make a latency prediction, *iPlane* simply adds the latencies associated with the individual path segments. Or, to predict the end-to-end bandwidth, *iPlane* computes the minimum of the bandwidth measured of each of the inter-cluster links along the predicted path, and the bandwidth of the client's access link, if available.

The rest of this section describes the techniques used to develop a functional *iPlane* that has wide coverage, incurs modest measurement load without unduly sacrificing coverage or detail, and uses topology structuring techniques to enable efficient measurement and accurate inference. The techniques are summarized in Table 1.

2.2 Mapping the Internet Topology

iPlane requires geographically distributed vantage points to map the Internet topology and obtain a collection of observed paths. PlanetLab servers, located at over 300 sites around the world, serve as the primary vantage points. We also enlist the use of public Looking Glass/Traceroute servers for low-intensity probing. Further, we are currently exploring the option of using data from DIMES [43], a system for aggregating low intensity measurements from normal PCs. Our primary tool for determining the Internet topology is traceroute, which allows us to identify the network interfaces on the forward path from the probing entity to the destination.

(On PlanetLab, we use an optimized version of the tool to reduce measurement load.) Determining what destinations to probe and how to convert the raw output of traceroute to a structured topology is nontrivial, an issue we address next.

2.2.1 Probe Target Selection

BGP snapshots, such as those collected by Route-Views [33], are a good source of probe targets. *iPlane* achieves wide coverage for the topology mapping process by obtaining the list of all globally routable prefixes in BGP snapshots, and choosing within each prefix a target .*I* address that responds to either ICMP or UDP probes. A .*I* address is typically a router and is hence more likely to respond to probes than arbitrary end-hosts.

To reduce measurement load, *iPlane* clusters IP prefixes into BGP atoms [4] for generating the target list. A BGP atom is a set of prefixes, each of which has the same AS path to it from any given vantage point. BGP atoms can be regarded as representing the knee of the curve with respect to measurement efficiency—probing within an atom might find new routes, but it is less likely to do so [4]. This task of determining a representative set of IP addresses is performed relatively infrequently, typically once every two weeks.

iPlane uses the PlanetLab nodes to perform exhaustive and periodic probing of the representative targets. In addition, iPlane schedules probes from public traceroute servers to a small random set of BGP atoms, typically making a few tens of measurements during the course of a day. The public traceroute servers serve as a valuable source of information regarding local routing policies. Note that in the long run, a functioning iPlane may actually serve to decrease the load on the public traceroute servers as iPlane, rather than the traceroute servers themselves, can be consulted for information on the Internet topology.

2.2.2 Clustering of Interfaces

Traceroute produces a list of network interfaces on the path from source to destination. However, interfaces on the same router, or in the same PoP, may have similar behavior. Hence, we partition network interfaces into "clusters" and use this more compact topology for more in-depth measurements and predictions. We define the clusters to include interfaces that are similar from a routing and performance perspective, i.e., interfaces belonging to the same PoP and interfaces within geographically nearby portions of the same AS [46]. Note that this clustering is performed on network interfaces in the Internet core, whereas the clustering of prefixes into BGP atoms was performed for end-host IP addresses. In fact, clustering addresses in the same prefix will be ineffective in the core as geographically distant interfaces are often assigned addresses in the same prefix.

First, iPlane identifies interfaces that belong to the same router. Interfaces that are potential alias candidates are identified using two different techniques. Employing the Mercator [19] technique, UDP probes are sent to a high-numbered port on every router interface observed in traceroutes. Interfaces that return responses with the same source address are considered as possible aliases. In addition, candidate alias pairs are also identified using the fact that interfaces on either end of a long-distance link are usually in the same /30 prefix. Candidate pairs that respond with similar IP-ID values to the UDP probes, and also respond with similar TTLs to the ICMP probes are deemed to be aliases. In one of our typical runs, of the 396,322 alias candidate pairs yielded by the Mercator technique, 340,580 pairs were determined to be aliases. The 918,619 additional alias candidate pairs obtained using the 130 heuristic yielded another 320,150 alias pairs.

Second, iPlane determines the DNS names assigned to as many network interfaces as possible. It then uses two sources of information - Rocketfuel's undns utility [47] and data from the Sarangworld project [40] – to determine the locations of these interfaces based on their DNS names. This step alone does not suffice for our purpose of clustering geographically co-located interfaces because: 1) several interfaces do not have a DNS name assigned to them, 2) rules for inferring the locations of all DNS names do not exist, and 3) incorrect locations are inferred for interfaces that have been misnamed. For IPs whose locations can be inferred from DNS names, the locations are validated by determining if they are consistent with the measured delays from traceroutes [28].

Third, to cluster interfaces for which a valid location was not determined, we develop an automated algorithm that clusters interfaces based on responses received from them when probed from a large number of vantage points. We probe all interfaces from all of iPlane's PlanetLab vantage points using ICMP ECHO probes. We use the TTL value in the response to estimate the number of hops on the reverse path back from every router to each of our vantage points. Our hypothesis is that routers in the same AS that are geographically nearby will have almost identical routing table entries and hence, take similar reverse paths back to each vantage point.

To translate this hypothesis into a clustering algorithm, each interface is associated with a reverse path length vector. This is a vector with as many components as the number of vantage points, and the i^{th} component is the length of the reverse path from the interface back to the i^{th} vantage point. We define the cluster distance between two vectors to be the L1 distance—the sum of the absolute differences between corresponding components, divided by the number of components. In our measurements, we have observed that the cluster distance between reverse path length vectors of co-located routers in an AS is normally less than 1.

Based on the metric discussed above, we can now present a technique for assigning interfaces without known locations to clusters. We start by initializing our clusters to contain those interfaces for which a location has been determined. Interfaces that have been determined to be co-located in an AS are in the same cluster. For each cluster, we compute the median reverse path length vector, whose i^{th} component is the median of the ith components of the vectors corresponding to all interfaces in the cluster. We then cluster all interfaces that do not belong to any cluster as follows. For each interface, we determine the cluster in the same AS as the interface, with whose median vector the interface's vector has the least cluster distance. If this minimum cluster distance is less than 1, the interface is added to the chosen cluster, otherwise a new singleton cluster is created. This clustering algorithm, when executed on a typical traceroute output, clusters 762,701 interfaces into 54,530 clusters. 653,455 interfaces are in 10,713 clusters of size greater than 10, while 21,217 interfaces are in singleton clusters.

2.3 Measuring the Internet Core

After clustering, iPlane can operate on a compact routing topology, where each *node* in the topology is a cluster of interfaces and each link connects two clusters. iPlane then seeks to determine a variety of link attributes that can be used to predict path performance. To achieve this goal, a centralized agent is used to distribute the measurement tasks such that each vantage point is assigned to repeatedly measure only a subset of the links. The centralized agent uses the compact routing topology to determine the assignments of measurement tasks to vantage points, communicates the assignment, and monitors the execution of the tasks. Only *iPlane* infrastructure nodes (namely, PlanetLab nodes) are used for these tasks.

2.3.1 Orchestrating the Measurement Tasks

There are three objectives to be satisfied in assigning measurement tasks to vantage points. First, we want to minimize the measurement load by measuring each link attribute from only a few vantage points (we employ more than one to correct for measurement noise). Second, the measurement should be load-balanced across all vantage points, *i.e.*, each vantage point should perform a similar number of measurements. Third, in order to measure the properties of each link as accurately as possible, we measure every link in the topology from the vantage point that is closest to it.

We have developed a novel "frontier" algorithm to perform the assignment of tasks to vantage points. The algorithm works by growing a frontier rooted at each vantage point and having each vantage point measure only those links that are at its frontier. The centralized agent performs a Breadth-First-Search (BFS) over the measured topology in parallel from each of the vantage points. Whenever a vantage point is taken up for consideration, the algorithm performs a single step of the BFS by following one of the traceroute paths originating at the vantage point. If it encounters a link whose measurement task has been assigned already to another vantage point, it continues the BFS exploration until it finds a new link that has not been seen before. This process continues until all the link measurements have been assigned to some vantage point in the system.

The centralized agent uses the above algorithm to determine the assignment of tasks and then ships the tasklist to the respective vantage points. Each target link is identified by the traceroute path that the vantage point can use to reach the link and by its position within the traceroute path. If a vantage point is no longer capable of routing to the link due to route changes, the vantage point reports this to the centralized agent, which in turn reassigns the task to a different vantage point.

Most link attributes, however, cannot be directly determined by the vantage points. For instance, when measuring loss rates, a vantage point can only measure the loss rate associated with the entire path from the vantage point to the target link; the loss rates of individual links have to be inferred as a post-processing operation. Once all vantage points report their measurements back to the centralized agent, the agent can perform the BFS style exploration of the topology to infer link properties in the correct order. For instance, assume that a vantage point v had probed the path v, \ldots, x, y and obtained a (oneway) loss rate measurement of $l_{v,y}$ for the entire path. The centralized agent can then infer the loss rate along the link (x, y) after inferring the loss rates for each of the links in v, \ldots, x , composing these individual loss rates to compute the loss rate $l_{v,x}$ along the segment $v \dots x$, and then calculating the loss rate for (x, y) using the equation

 $(1-l_{v,y}) = (1-l_{v,x}) \cdot (1-l_{x,y})$. Since the link property inference is performed as a BFS traversal, we are guaranteed that loss rates for all the links along v, \ldots, x have been inferred before we consider the link (x, y).

In our current system, the centralized agent schedules and monitors roughly 2700K measurements per day, a management load that a single centralized agent can easily bear. Fault tolerance is an issue, but is addressed by a simple failover mechanism to a standby controller. Note that the processed data is served to applications from a replicated database to ensure high availability.

2.3.2 Measurement of Link Attributes

We next outline the details of the loss rate, bottleneck capacity and available bandwidth measurements performed from each vantage point. Previous research efforts have proposed specific ways to measure each of these properties; our goal is to integrate these techniques into a useful prediction system. Latencies of path segments can be derived directly from the traceroute data gathered while mapping the topology, and therefore do not need to be measured explicitly.

Loss Rate Measurements: We perform loss rate measurements along path segments from vantage points to routers in the core by sending out probes and determining the fraction of probes for which we get responses. We currently use the simple method of sending TTLlimited singleton ICMP probes with a 1000-byte payload. When the probe's TTL value expires at the target router, it responds with a ICMP error message, typically with a small payload. When a response is not received, one cannot determine whether the probe or the response was lost, but there is some evidence from previous studies that small packets are more likely to be preserved even when routers are congested [32]. We therefore currently attribute all of the packet loss to the forward path; the development of more accurate techniques is part of ongoing work.

Capacity Measurements: We perform capacity measurements using algorithms initially proposed by Bellovin [3] and Jacobson [24] that vary the packet size and determine the delay induced by increased packet sizes. For each packet size, a number of probes (typically 30-40) of that size are sent to an intermediate router and the minimum round-trip time is noted. The minimum round-trip time observed over many probes can be regarded as a baseline path latency measurement with minimal queueing delays. By performing this experiment for different packet sizes, one can determine the increased transmission cost per byte. When this experiment is performed for a sequence of network links in succession, the capacity of each link can be determined. Note that our capacity measurements may underestimate a cluster link if it consists of multiple parallel physical links.

Available Bandwidth Measurements: Once we have link capacities, we can probe for available bandwidth along path segments using packet dispersion techniques such as Spruce [50], IGI [21], Pathload [25]. A simple measurement is performed by sending a few, equally spaced, short probes at the believed bottleneck capacity of the path segment, and then measuring how much delay they induce. The slope of the delay increase will indicate how much background traffic arrived during the same time period as the probe. For instance, if the probes are generated with a gap of Δ_{in} through a path segment of capacity C and if the measured gap between between the probe replies is Δ_{out} , one can estimate the available bandwidth as $C \cdot (1 - \frac{\Delta_{out} - \Delta_{in}}{\Delta_{in}})$. An important detail is that the packets have to be scheduled at the desired spacing, or else the measurement is not valid. Fortunately, even on heavily loaded PlanetLab nodes, it is possible to realize the desired scheduling most of the time.

2.4 Opportunistic Edge Measurements

To provide a comprehensive data set on which to infer current properties of paths to end-hosts, it is necessary for iPlane to maintain an up-to-date map of the network that extends to the very edge. However, the measurement techniques outlined above are unlikely to work as they, as most other active measurements, require end-hosts to respond to unsolicited ICMP, UDP or TCP packet probes. Also, measurements to end-hosts are frequently misinterpreted by intrusion detection systems as attacks. Hence, we pursue an opportunistic approach to data collectionmeasuring paths to end-hosts while interacting with them over normal connections. We participate in the popular file-distribution application BitTorrent [9] and gather measurements from our exchanges with the peers in this swarming system. Note that BitTorrent has the further desirable property that anyone can connect to anyone, allowing us to arrange measurements of multiple paths to participating edge hosts.

BitTorrent is used daily by thousands of end users to distribute large files. BitTorrent is one example of a large class of *swarming* data distribution tools. By participating in several BitTorrent swarms, we have the opportunity to interact with a large pool of end-hosts. We measure properties of the paths to peers while exchanging data with them as part of the swarming system.

We currently gather two kinds of measurements using our opportunistic measurement infrastructure.

 Packet traces of TCP flows to end-hosts. These traces provide information about packet inter-arrival times, loss rates, TCP retransmissions and round trip times.
 We use the inter-arrival times between data packets to measure bottleneck bandwidth capacities of paths from clients to vantage points, as described further in Section 3.

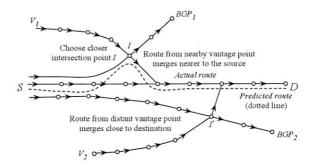


Figure 1: The path from S to D is obtained by composing a path from S with a path going into D from a vantage point close to S (V_1). BGP_1 and BGP_2 are destinations in two random prefixes to which S performs traceroutes.

 Traceroutes to end-hosts. When a peer connects to our measurement node, we conduct a traceroute to that host. We record this data and add it to our atlas.

2.5 Performance Prediction

Next, we describe how to predict path properties between an arbitrary pair of nodes based on the above measurements. The prediction proceeds in two steps. First, we predict the forward and reverse paths connecting the two nodes. Second, we aggregate measured link-level properties to predict end-to-end path properties.

Path Prediction We use a technique we earlier developed [31] based on composing observed path segments to predict unknown paths. Consider a source S and destination D. If S is a vantage point, then we simply return the measured path from S to D. Else, we determine an appropriate intersection point I in the measured subgraph of the Internet such that—(a) the AS hop count of the path S.I.D is minimum, and (b) the latency from Sto the point where the path S.I.D exits the first-hop AS is minimum, in that order (Figure 1). The underlying principle is similarity of routes, i.e., with a sufficiently large number of vantage points, the path to a destination (D) from any node (S) will be similar to the path from a vantage point or router (I) located nearby. Condition (a)encodes the default path selection criterion used by BGP in the absence of conflicting local preference policies. Condition (b) encodes the default early exit intradomain routing policy. Note that the above technique is guaranteed to return a path (albeit an inflated one), since every path of the form S.V.D, for each vantage point V, belongs to the measured subgraph.

As we noted earlier, we make measurements to BGP atoms rather than to all destinations. In [31], we note that adding a small number of measured paths originating from the client S significantly improves the prediction accuracy for paths sourced at S. Using these measurements, the path from S to D is S to I to D's atom to D. If there is a measurement of the last hop from D's atom to D, we use it; otherwise, we estimate it using

a representative node in the atom (e.g., from BitTorrent measurements). Briefly summarizing the main results from [31], we can predict the AS path exactly right for around 70% of the paths evaluated, and the latency estimates obtained using this model were significantly better than those yielded by Vivaldi [11], a popular network coordinate system.

Path Properties Given predicted paths as above, we can estimate end-to-end properties by aggregating linklevel properties. For example, we predict TCP transfer time using widely accepted models [35, 5]. For this, we separately predict the forward and reverse paths between the source and the destination. The latency on the forward path S.I.D-atom.D is estimated as the sum of the latency estimates for each segment. We similarly estimate the latency along the reverse path, and then compute the RTT between the two end-hosts to be the sum of our latency estimates along the forward and reverse paths. The loss rate on the predicted forward path is estimated from the probability of a loss on any of its constituent links while bandwidth is the minimum value across the links. The access link capacities of these endhosts, if available based on BitTorrent measurements to hosts in the same /24 prefixes, are also used to estimate the end-to-end bottleneck bandwidth.

Recently, He et al. [20] argued that the best way to accurately predict TCP throughput is to send TCP flows and use history-based predictors. Although we have not implemented these, our use of passive BitTorrent logs is amenable to incorporating such predictors.

2.6 Securing iPlane

iPlane allows untrusted users to contribute measurements, so it is vulnerable to attacks aimed at polluting its information. For instance, a client can claim to have better connectivity than actuality in order to improve its position within an overlay service that uses *iPlane*. *iPlane* reduces this risk by using client data only for those queries issued by the same client; falsified measurements will not affect the queries issued by other clients.

We do however trust traceroute servers to provide unbiased data, though the traceroute servers are not under our control. An ISP hosting a traceroute server might bias its replies from the server to better position its clients, for example, to attract more BitTorrent traffic and thereby generate more revenue. We have the ability to use verification to address this – compare the results from multiple vantage points for consistency – but have not implemented it yet.

2.7 Query Interface

The query interface exported by *iPlane* must be carefully designed to enable a diverse range of applications. Our current implementation of the query interface ex-

poses a database-like view of path properties between every pair of end-hosts in the Internet. For every source-destination pair, there exists a row in the view with *iPlane*'s predicted path between these hosts and the predicted latency, loss rate, and available bandwidth along this path. Any query to *iPlane* involves an SQL-like query on this view – selecting some rows and columns, joining the view with itself, sorting rows based on values in certain columns, and so on. The database view is merely an abstraction. *iPlane* does not compute apriori the entire table comprising predictions for every source-destination pair; instead it derives necessary table entries on-demand.

For example, a CDN can determine the closest replica to a given client by selecting those rows that predict the performance between the client and any of the CDN's replicas. A suitable replica can then be determined by sorting these rows based on a desired performance metric. To choose a good detour node for two end-hosts to conduct VoIP, the rows predicting performance from the given source can be joined with the set of rows predicting performance for the given destination. A good detour is one that occurs as the destination in the first view and as the source in the second view, such that the composed performance metrics from these rows is the best. These queries can be invoked in any one of the following ways.

Download the Internet Map: We have implemented a library that provides an interface to download the current snapshot of the entire annotated Internet map or a geographic region, to process the annotated map, and to export the above SQL-like view. An application simply links against and invokes the library locally.

On-the-fly Queries: Applications that do not wish to incur the costs of downloading the annotated map and keeping it up-to-date, can query a remote iPlane service node using non-local RPCs. Note that clients of CDNs, such as Akamai and Coral, typically tolerate some indirection overhead in determining the nearest replica. To support such applications, iPlane downloads the annotated map of the Internet to every PlanetLab site, and then provides an RPC interface to the data. Further, as some applications might need to make multiple back-toback queries to process iPlane's measurements, we assist the application in lowering its overheads by allowing it to upload a script that can make multiple local invocations of iPlane's library. The current implementation requires that this script be written in Ruby, as Ruby scripts can be executed in a sandboxed environment and with bounded resources [49]. The output of the script's execution is returned as the response to the RPC.

Network Newspaper: Apart from downloading the Internet graph and issuing on-the-fly queries, a third model that we plan to support is a publish-subscribe interface that allows users to register for information up-

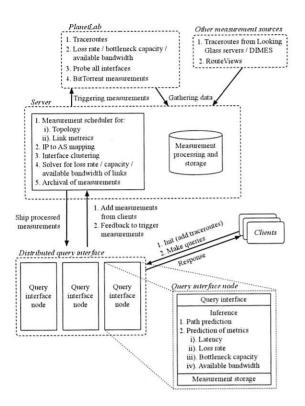


Figure 2: Overall architecture of iPlane.

dates about specific portions of the Internet graph. This interface allows users to subscribe to their "view" of the Internet, i.e., all paths originating from a user to all BGP atoms, or insert triggers to be notified of specific events, e.g., when a critical link fails. Implementing such an interface is part of our future work.

The various components in our current implementation of *iPlane*, and the interaction between these components is depicted in Figure 2.

3 System Setup and Evaluation

In this section, we present details of our deployment of *iPlane*. We provide an overview of the measurements we conducted as part of our deployment. We also outline the tests we conducted to validate our measurements. All of our validation is performed on paths between PlanetLab nodes; our goal in the future is to use client measurements (e.g., DIMES [43]) to broaden the validation set. The number of PlanetLab nodes used varies with each experiment because of the variable availability of some nodes.

3.1 Measuring the Core

We first consider results from a typical run of our mapping process. We performed traceroutes from PlanetLab nodes in 163 distinct sites. The targets for our traceroutes were .1 addresses in each of 91,498 prefixes determined from the RouteViews BGP snapshot, though mea-

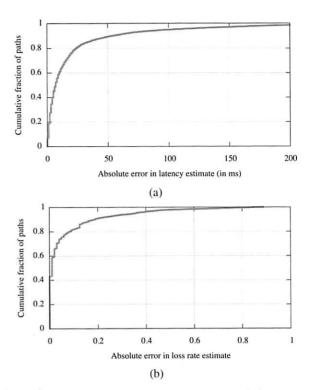


Figure 3: Distribution of errors in (a) latency, and (b) loss rate estimation.

suring paths to one address in each BGP atom should suffice. We probed all interfaces observed in our measured topology with UDP and ICMP probes, and clustered the interfaces based on their responses.

Once a map of the Internet's core was gathered, we employed our "frontier" BFS algorithm to determine paths to be probed from each of the 385 PlanetLab nodes present at the 163 sites used; for link metrics, we use multiple nodes per site. To determine the properties of 270,314 inter-cluster links seen in our measured topology, each vantage point was assigned to measure only around 700 paths. Loss rate, capacity, and available bandwidth were measured for each of the assigned paths. These measurements were then processed to determine properties for every cluster-level link in our measured topology.

To validate the predictive accuracy of *iPlane*, we compared properties of paths between PlanetLab nodes with the corresponding values predicted by *iPlane*. We measured the latency and loss rate along every path beween any two PlanetLab nodes. To predict the performance, we assume that we have the probe information collected by the other 161 sites, excluding the source and destination under consideration. We then added 10 traceroutes from the source and destination to random nodes to simulate the behavior of participating clients. Each experiment was performed independently to ensure no mixing of the measurement and validation set. Figure 3 compares the latency and loss rate estimates made by

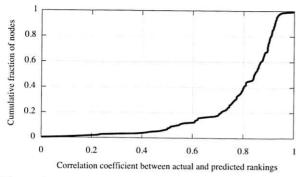


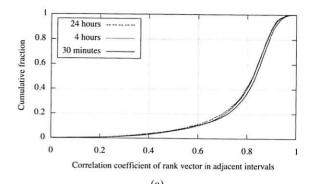
Figure 4: Rank correlation coefficient between actual and predicted TCP throughput.

iPlane with the true values. For 77% of paths, *iPlane*'s latency estimates have error less than 20ms, and for 82% of paths, loss rate estimates have error less than 10%.

Further, we evaluated how predictive of path performance are *iPlane*'s estimates of latency and loss rate in combination. The desired property of these estimates is that they help distinguish between paths with good and bad performance. We compared the order of paths from each PlanetLab node in terms of actual and predicted performance. For each node, we ranked all other nodes in terms of TCP throughput, considering throughput to be inversely proportional to latency and the square root of loss rate [35]. These rankings were computed independently using measured path properties and using *iPlane*'s predictions for these properties. Figure 4 plots the correlation coefficient between the actual and *iPlane* predicted rankings across all PlanetLab nodes. For 80% of the nodes, the correlation coefficient is greater than 0.7.

3.2 Scalability

We now discuss the measurement load required to generate and maintain a frequently refreshed map of the Internet. The measurement tasks performed by iPlane have two primary objectives—mapping of the Internet's cluster-level topology and determination of the properties of each link in the measured topology. Measurement of link properties incurs higher measurement overhead when compared to the probe traffic needed to perform a traceroute, but scales better. With more vantage points, the topology discovery traffic per node remains the same, but the overhead per node for measuring link metrics scales down, allowing the same fidelity for less overhead per node. The measurement load associated with each technique in our measurement apparatus is summarized in Table 2. These numbers assume the availability of 400 PlanetLab nodes at 200 sites. Our main result is that iPlane can produce an updated map of the Internet's routing topology every day with as little as 10Kbps of probe traffic per vantage point, and update the map of link-level attributes once every 6 hours with around 100Kbps of probe traffic per vantage point, suggesting that iPlane can refresh the Internet map frequently.



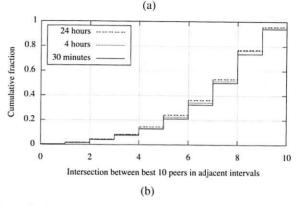


Figure 5: Stationarity of measurements over different intervals over the course of a day.

3.3 Stationarity of Measurements

iPlane's measurements change over time with changes in the routes in the Internet and the traffic they carry. We again use PlanetLab data to estimate whether it suffices for *iPlane* to update its map every 6 hours. We are currently in the process of evaluating the stationarity of path properties for non-PlanetLab destinations as well.

Over a period of 2 days, we measured the latency and loss rate between PlanetLab nodes once every 30 minutes. For this study, we used a dataset of 174 PlanetLab sites spanning 29 countries. In every interval, we computed for each node the ranking of all other nodes in terms of TCP throughput. To evaluate the flux in path properties over a 30 minute timescale, we compared these rankings between adjacent 30 minute intervals. For each PlanetLab node, we computed the correlation coefficient between the ranking vectors from adjacent intervals as well as computed the intersection between the top 10 nodes in these ranking vectors. To compare this with the flux in measurements over longer timescales, we also performed these computations across intervals 1 hour, 2 hours, 4 hours, 8 hours, 16 hours and 24 hours apart.

Figure 5(a) shows that the median correlation coefficient between the rankings is greater than 0.8 across all intervals from 30 minutes to a day. Similarly, Figure 5(b) shows that in the median case, 7 of the top 10 nodes in this ranking are identical on timescales from 30 minutes to a day. Though these results are only for paths between

Measurement Task	Tool / Technique	Frequency	Probing rate / node
Topology Mapping	traceroute	Once a day	200 vantage points × 50K atoms — 2.5Kbps
Clustering	UDP probes for source-address-based alias resolution, ICMP-ECHO probes for RTTs and reverse TTLs	One day every week	100 vantage points × 800K interfaces — 6Kbps
Capacity measurements	"frontier" algorithm applied to cluster-level topology for path assignment, <i>pathchar</i> for bandwidth capacity	Once a day	400 vantage points × 700 links — 13Kbps
Loss rate and available bandwidth measurements	"frontier" algorithm for path assignment, TTL-limited probes for loss rate, <i>spruce</i> for available bandwidth	Continuous (every 6 hours)	400 vantage points × 700 links — 80Kbps

Table 2: Complexity of measurements techniques used in *iPlane* based on the following assumptions. A UDP/ICMP probe is 40 bytes. A traceroute incurs a total of 500B on average. The per-link loss rate, available bandwidth, and capacity measurements require 200KB, 100KB, and 200KB of probe traffic respectively.

PlanetLab nodes, they seem to indicate that there is little value in updating the map more frequently than once every few hours, compared to once every 30 minutes.

3.4 Measurements to End-Hosts

To measure the edges of the Internet, we deployed a modified BitTorrent client on 367 PlanetLab nodes. As described in Section 2.4, our infrastructure for measuring the edge involves the millions of users who frequently participate in the BitTorrent filesharing application. Every hour, we crawl well-known public websites that provide links to several thousand .torrent files to put together a list of 120 popular swarms. The number of swarms for consideration was chosen so as to ensure the participation of several of our measurement vantage points in each swarm. The number of PlanetLab nodes designated to a swarm is proportional to the number of peers participating in it.

Each PlanetLab node runs a BitTorrent client that we have modified in several ways to aid in our measurements. First, the modified client does not upload any data nor does it write any data that it downloads onto disk. Second, our client severs connections once we have exchanged 1MB of data, which suffices for purposes of our measurements. Finally, we introduce a shadow tracker a database that coordinates measurements among all PlanetLab nodes participating in a single swarm. Instead of operating only on the set of peers returned by the original tracker for the swarm, our modified client also makes use of the set of peers returned to any measurement node. Clients preferentially attempt to connect and download data from peers that have not yet been measured by a sufficient number of vantage points. These modifications are crucial for measurement efficiency and diversity since typical BitTorrent trackers permit requesting only a restricted set (50-100) of participating peers once every 30 minutes or more. Such short lists are quickly exhausted by our modified client.

During a 48 hour period, our measurement nodes connected to 301,595 distinct IP addresses, and downloaded sufficient data to measure the upload bandwidth capacity

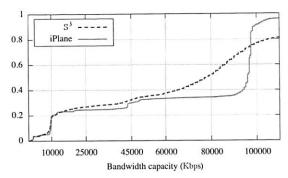


Figure 6: CDFs of estimated bandwidth capacity on paths between PlanetLab nodes as measured by iPlane and S^3 .

from 70,428. These hosts span 3591 distinct ASs, 19,639 distinct BGP prefixes, and 160 different countries.

3.5 Validation of BitTorrent capacity measurements

Our edge bandwidth capacity measurement relies on inter-arrival times observed between data packets in the connections we maintain with BitTorrent peers. We implemented the multiQ [27] technique to infer end-to-end bottleneck bandwidth capacity from these inter-arrival times. Although the accuracy of multiQ presented in previous studies is encouraging, the unique properties of PlanetLab motivated us to provide further validation. To verify that multiQ yields reasonable data with short TCP traces in the presence of cross traffic on machines under heavy load, we compared our measurements with those made by S^3 [13].

We setup a test torrent and had our measurement clients running on 357 PlanetLab nodes participate in this torrent. From this setup, we opportunistically measured the bottleneck bandwidth capacities between these PlanetLab nodes. The dataset we gathered from this experiment had 10,879 paths in common with measurements made by S^3 on the same day. Figure 6 compares the bandwidth capacities measured by the two methods. The measurements made by iPlane closely match those of S^3 for capacities less than 10 Mbps. At higher bandwidth capacities, they are only roughly correlated. We attribute

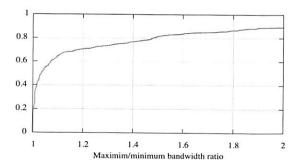


Figure 7: CDF of the ratio of maximum to minimum measured bandwidth capacity for /24 address prefixes with multiple measurements from the same vantage point across time.

this difference to the use of user-level timestamps by S^3 . As inter-packet spacing can be rather small for high capacity paths, user-level timestamps are likely to be inaccurate in the highly loaded PlanetLab environment. Our measurement setup makes use of kernel-level timestamps and is therefore less sensitive to high CPU load. For typical access link bandwidths, the two tools produce similar data; the value of using BitTorrent is that it works with unmodified clients that sit behind firewalls or NATs that would drop active measurement probes. The more discernible steps in the iPlane line in Figure 6 are at 10Mbps, 45Mbps (T3), and 100Mbps, which correspond to typical ISP bandwidth classes.

3.6 Clustering of end-hosts

Although the data produced by our opportunistic strategy is extensive, it is by no means complete. Not every client participates in popular torrents. In Figure 7, we explore the validity of using BitTorrent measurements to predict the performance of other clients in the same prefix. For every /24 prefix in which we have measurements to multiple end-hosts from the same vantage point, we compute the ratio of the maximum to the minimum measured bandwidth capacity. For 70% of /24 prefixes, the capacities measured differ by less than 20%.

4 Application Case Studies

In this section, we show how applications can benefit from using *iPlane*. We evaluate three distributed services for potential performance benefits from using *iPlane*.

4.1 Content Distribution Network

Content distribution networks (CDNs) such as Akamai, CoDeeN and Coral [1, 52, 16] redirect clients to a nearby replica. The underlying assumption is that distance determines network performance. However, there is more to network performance than just distance, or round trip time. TCP throughput, for example, depends on both distance and loss rate [35, 5]. Even for small web documents, loss of a SYN or a packet during slow start can markedly inflate transfer time. A CDN using *iPlane* can track the RTT, loss rate, and bottleneck capacity from

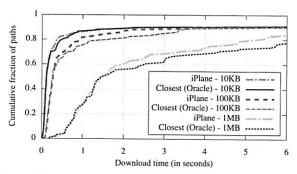


Figure 8: CDF of download times from replicas in the CDN chosen by the *iPlane* and from replicas closest in terms of latency. Each download time is the median of 5 measurements.

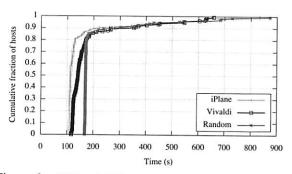


Figure 9: CDFs of BitTorrent download completion times with and without informed peer selection at the tracker.

each replica to the rest of the Internet. The CDN can then arrange for its name servers to redirect the client to optimize using the model of its choice.

We emulate a small CDN comprising 30 randomly chosen PlanetLab nodes. Each node serves 3 files of sizes 10KB, 100KB and 1MB. We use 141 other Planet-Lab nodes to emulate clients. Each client downloads all 3 files from the replica that provides the best TCP throughput as predicted by the PFTK model [35] using iPlane's estimates of RTT and loss rate, and from the replica closest in terms of actual measured RTT. Note that this comparison is against an optimum that cannot be achieved without extensive probing. A real CDN will only have estimated RTTs available. Figure 8 compares the download times experienced by the clients in either case, excluding the latency of redirecting to the replica. Choosing the replica for optimized TCP throughput based on iPlane's predictions provides slightly better performance than choosing the closest replica. Though these results are only indicative, they suggest that iPlane with its ability to provide multi-attribute network performance data will be more effective than systems such as OASIS [17] that simply optimize for RTT.

4.2 BitTorrent

We next show how *iPlane* can enable informed peer selection in popular swarming systems like BitTorrent. In current implementations, a centralized BitTorrent *tracker*

serves each client a random list of peers. Each client enforces a tit-for-tat bandwidth reciprocity mechanism that incents users to contribute more upload bandwidth to obtain faster downloads. However, the same mechanism also serves to optimize path selection at a local levelpeers simply try uploading to many random peers and eventually settle on a set that maximizes their download rate. Because reasoning about peer quality occurs locally at each client, each client needs to keep a large pool of directly connected peers (60-100 for typical swarms) even though at any time only a few of these (10-20) are actively engaged in data transfer with the client. This overhead and consequent delayed convergence is fundamental: with only local information, peers cannot reason about the value of neighbors without actively exchanging data with them. iPlane's predictions can overcome the lack of prior information regarding peer performance and can thus enable a clean separation of the path selection policy from the incentive mechanism.

We built a modified tracker that uses *iPlane* for informed peer selection. Instead of returning random peers, the tracker uses the *iPlane*'s loss rate and latency estimates to infer TCP throughput. It then returns a set of peers, half of which have high predicted throughput and the rest randomly selected. The random subset is included to prevent the overlay from becoming disconnected (e.g., no US node preferring a peer in Asia).

We used our modified tracker to coordinate the distribution of a 50 megabyte file over 150 PlanetLab nodes. We measured the time taken by each of the peers to download the file after the seed was started. Figure 9 compares the download times observed with *iPlane* predictions against those of peerings induced by Vivaldi coordinates [11] and an unmodified tracker. Informed peer selection causes roughly 50% of peers to have significantly lower download times.

Although preliminary, these performance numbers are encouraging. We believe that better use of information from the *iPlane* can lead to even further improvements in performance. Our selection of 50% as the fraction of random peers was arbitrary, and we are currently investigating the tradeoff between robustness and performance, as well as the degree to which these results extend to swarms with a more typical distribution of bandwidths.

4.3 Voice Over IP

Voice over IP (VoIP) is a rapidly growing application that requires paths with low latency, loss and jitter for good performance. Several VoIP implementations such as Skype [45] require relay nodes to connect end-hosts behind NATs/firewalls. Choosing the right relay node is crucial to providing acceptable user-perceived performance [39]. Reducing end-to-end latency is important since humans are sensitive to delays above a thresh-

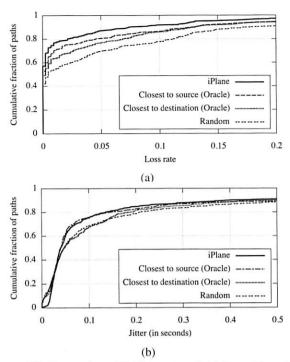


Figure 10: Comparison of (a) loss rate and (b) jitter with and without use of *iPlane* for end-to-end VoIP paths.

old. Low loss rates improve sound quality and reduce throughput consumed by compensating codecs. Measures of user-perceived sound quality such as *mean opinion score* [51] have been shown to be highly correlated with loss rate and end-to-end delay. Thus, VoIP applications can benefit from *iPlane*'s predictions of latency and loss rate in choosing the best possible relay node.

To evaluate *iPlane*'s ability to successfully pick good relay nodes, we emulated VoIP traffic patterns on PlanetLab. We considered 384 pairs of PlanetLab nodes, chosen at random, as being representative of end-hosts participating in a VoIP call. Between each pair, we emulated a call by sending a 10KBps UDP packet stream via another PlanetLab node chosen as the relay node. We tried 4 different relay options for each pair chosen based on (i) the *iPlane*'s estimates of latency and loss rate, (ii) latency to the source, (iii) latency to the destination, and (iv) random choice. The *iPlane*-informed choice was obtained by first querying for the 10 relay options that minimize end-to-end loss and then, choosing the one that minimized end-to-end delay among these options.

Each emulated call lasted for 60 seconds, and the end-to-end loss rate and latency were measured. Figure 10(a) shows that significantly lower loss rates were observed along relay paths chosen based on *iPlane*'s predictions. Additionally, Figure 10(b) shows that *iPlane* also helps to reduce jitter, which we computed as the standard deviation of end-to-end latency. These results demonstrate the potential for the use of *iPlane* in VoIP applications.

5 Related Work

iPlane bridges and builds upon ideas from network measurement, performance modeling, Internet tomography, and recent efforts towards building a knowledge plane for the Internet. We believe that an Internet-scale instantiation of *iPlane* is greater than the sum of its parts, and relate individual contributions to prior work.

Information Plane Clark et al. [8] pioneered the broad vision of a knowledge plane to build large-scale, self-managing and self-diagnosing networks based on tools from AI and cognitive science. Several research efforts have since addressed pieces of this problem.

Several efforts have looked at monitoring end-host performance and in optimizing the query processing engine of the information plane. Examples include Sophia [53], PIER [23], and IrisNet [18]. The above systems have a different focus than ours. They manage information about nodes (e.g., PlanetLab nodes, routers in an ISP, or sensors) under control of the information plane. We target predictions of path performance at Internet-scale.

Link Metrics IDMaps [15] is an early example of a network information service that estimates the latency between an arbitrary pair of nodes using a small set of vantage points as landmarks. Subsequently, Ng and Zhang [34] discovered that Internet distances can be embedded on to a low-dimensional Euclidean space. Such embeddings can be used to predict latencies between a large number of nodes by measuring latencies from a small number of vantage points to these nodes-a methodology refined by several others [54, 10, 11, 44]. A key limitation of these techniques is that they treat the Internet as a black box and are only predictive, i.e., they do not explain why, if at all, their predictions are correct. As a result, they have serious systematic deficiencies, e.g., a significant fraction of Internet paths are known to have detours [41], however, metric embeddings obey the triangle inequality and will predict no detours.

Our previous work on a structural technique [31] to predict Internet paths and latencies, and experiences reported by independent research groups with respect to latency prediction [17], available bandwidth estimation [22], and the practical utility of embedding techniques [30] echo the need for structural approaches to predict sophisticated path metrics.

Inference Techniques Chen et al. [7] proposed an algebraic approach to infer loss rates on paths between all pairs of nodes based on measured loss rates on a subset of the paths. Duffield et al. [14] proposed a multicast-based approach to infer link loss rates by observing loss correlations between receivers. Rocketfuel [47] estimates ISP topologies by performing traceroutes from a set of vantage points, while the Doubletree [12] system efficiently prunes redundant traceroutes. Our frontier (Section 2.3)

algorithm to efficiently target specific links for measurement is similar in spirit.

Passive Measurements Padmanabhan et al. [36] and Seshan et al.[42] propose passive measurements at Web servers and end-hosts respectively to predict path metrics. PlanetSeer by Zhang et al. [55] is a failure monitoring system that uses passive measurements at CDN caches under their control to diagnose path failures postmortem. Jaiswal et al. [26] propose a "measurements-inthe-middle" technique to infer end-to-end path properties using passive measurements conducted at a router. In contrast to these systems that perform passive measurements of existing connections, we participate in BitTorrent swarms and opportunistically create connections to existing peers for the explicit purpose of observing network behavior. Previously, opportunistic measurements have relied on sprurious traffic in the Internet [6]. iPlane could also validate and incorporate measurement data from passive measurement sources, such as widely deployed CDNs, and such integration is part of future work.

6 Conclusion

The performance and robustness of overlay services critically depends on the choice of end-to-end paths used as overlay links. Today, overlay services face a tension between minimizing redundant probe overhead and selecting good overlay links. More importantly, they lack an accurate methods to infer path properties between an arbitrary pair of end-hosts. In this paper, we showed that it is possible to accurately infer sophisticated path properties between an arbitrary pair of nodes using a small number of vantage points and existing infrastructure. The key insight is to systematically exploit the Internet's structural properties. Based on this observation, we built the iPlane service and showed that it is feasible to infer a richly annotated link-level map of the Internet's routing topology once every few hours. Our case studies suggest that iPlane can serve as a common information plane for a wide range of distributed services such as content distribution, file swarming, and VoIP.

Acknowledgments

We would like to thank Jay Lepreau, Ratul Mahajan, KyoungSoo Park, Rob Ricci, Neil Spring and the anonymous OSDI reviewers for their valuable feedback on earlier versions of this paper. We also thank Peter Druschel for serving as our shepherd. This research was partially supported by the National Science Foundation under Grants CNS-0435065 and CNS-0519696.

References

- [1] Akamai, Inc. home page. http://www.akamai.com.
- [2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In SOSP, 2001.
- [3] S. Bellovin. A best-case network performance model. Technical report, ATT Research, 1992.

- [4] A. Broido and kc claffy. Analysis of routeViews BGP data: policy atoms. In Network Resource Data Management Workshop, 2001.
- [5] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *INFOCOM*, 2000.
- [6] M. Casado, T. Garfinkel, W. Cui, V. Paxson, and S. Savage. Opportunistic measurement: Extracting insight from spurious traffic. In *HotNets*, 2005.
- [7] Y. Chen, D. Bindel, H. Song, and R. H. Katz. An algebraic approach to practical and scalable overlay network monitoring. In SIGCOMM, 2004.
- [8] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski. A knowledge plane for the Internet. In SIG-COMM, 2003.
- [9] B. Cohen. Incentives build robustness in BitTorrent. In *P2PEcon*, 2003.
- [10] M. Costa, M. Castro, A. Rowstron, and P. Key. PIC: Practical Internet coordinates for distance estimation. In ICDCS, 2004.
- [11] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In SIGCOMM, 2004
- [12] B. Donnet, P. Raoult, T. Friedman, and M. Crovella. Efficient algorithms for large-scale topology discovery. In SIGMETRICS, 2005.
- [13] C. Dovrolis, P. Ramanathan, and D. Moore. Packet dispersion techniques and a capacity estimation methodology. *IEEE/ACM Transactions on Networking*, 2004.
- [14] N. G. Duffield, F. L. Presti, V. Paxson, and D. F. Towsley. Inferring link loss using striped unicast probes. In *INFO-COM*, 2001.
- [15] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A global Internet host distance estimation service. *IEEE/ACM Transactions on Networking*, 2001
- [16] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In NSDI, 2004.
- [17] M. J. Freedman, K. Lakshminarayanan, and D. Mazieres. OASIS: Anycast for any service. In NSDI, 2006.
- [18] P. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Iris-Net: An architecture for a world-wide sensor web. *IEEE Pervasive Computing*, 2(4), 2003.
- Pervasive Computing, 2(4), 2003.
 [19] R. Govindan and H. Tangmunarunkit. Heuristics for Internet map discovery. In INFOCOM, 2000.
- [20] Q. He, C. Dovrolis, and M. Ammar. On the predictability of large transfer TCP throughput. In SIGCOMM, 2005.
- [21] N. Hu and P. Steenkiste. Evaluation and characterization of available bandwidth probing techniques. *IEEE JSAC*, 21(6), 2003.
- [22] N. Hu and P. Steenkiste. Exploiting Internet route sharing for large scale available bandwidth estimation. In *IMC*, 2005.
- [23] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In VLDB, 2003.
- [24] V. Jacobson. Pathchar. ftp://ftp.ee.lbl.gov/pathchar.
- [25] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. In SIGCOMM, 2002.
- [26] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Formal analysis of passive measurement inference techniques. In *INFOCOM*, 2006.
- ference techniques. In *INFOCOM*, 2006.

 [27] S. Katti, D. Katabi, C. Blake, E. Kohler, and J. Strauss. MultiQ: Automated detection of multiple bottleneck capacities along a path. In *IMC*, 2004.
- [28] E. Katz-Bassett, J. P. John, A. Krishnamurthy, D. Wetherall, T. Anderson, and Y. Chawathe. Towards IP geolocation using delay and topology measurements. In *IMC*, 2006.
- [29] S. Lee, P. Sharma, S. Banerjee, S. Basu, and R. Fonseca. Measuring bandwidth between planetlab nodes. In PAM, 2005.

- [30] E. K. Lua, T. Griffin, M. Pias, H. Zheng, and J. Crowcroft. On the accuracy of embeddings for Internet coordinate systems. In *IMC*, 2005.
- [31] H. V. Madhyastha, T. E. Anderson, A. Krishnamurthy, N. Spring, and A. Venkataramani. A structural approach to latency prediction. In *IMC*, 2006.
- [32] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In SOSP, 2003.
- [33] D. Meyer. RouteViews. http://www.routeviews.org.
- [34] T. S. E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *INFOCOM*, 2002.
- [35] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. CCR, 28(4), 1998.
- [36] V. N. Padmanabhan, L. Qiu, and H. J. Wang. Passive network tomography using bayesian inference. In *IMW*, 2002.
- [37] A. Parker. CacheLogic. http://www.cachelogic.com/ research/slide1.php.
- [38] L. Peterson. Personal communication.
- [39] S. Ren, L. Guo, and X. Zhang. ASAP: an AS-aware peerrelay protocol for high quality VoIP. In *ICDCS*, 2006.
- [40] Sarangworld project. http://www.sarangworld.com/TRACEROUTE/.
- [41] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: a case for informed Internet routing and transport. *IEEE Micro*, 19(1), 1999.
- [42] S. Seshan, M. Stemm, and R. Katz. SPAND: Shared passive network performance discovery. In USITS, 1997.
- [43] Y. Shavitt and E. Shir. DIMES: Let the Internet measure itself. CCR, 35(5), 2005.
- [44] Y. Shavitt and T. Tankel. On the curvature of the Internet and its usage for overlay construction and distance estimation. In *INFOCOM*, 2004.
- [45] Skype home page. http://www.skype.com.
- [46] N. Spring, R. Mahajan, and T. Anderson. Quantifying the causes of path inflation. In SIGCOMM, 2003.
- [47] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with Rocketfuel. *IEEE/ACM Transactions on Networking*, 2004.
- [48] N. Spring, D. Wetherall, and T. Anderson. Reverseengineering the Internet. In *HotNets*, 2003.
- [49] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public Internet measurement facility. In *USITS*, 2003.
 [50] J. Strauss, D. Katabi, and F. Kaashoek. A measurement
- [50] J. Strauss, D. Katabi, and F. Kaashoek. A measurement study of available bandwidth estimation tools. In *IMC*, 2003.
- [51] S. Tao, K. Xu, A. Estepa, T. Fei, L. Gao, R. Guerin, J. Kurose, D. Towsley, and Z.-L. Zhang. Improving VoIP quality through path switching. In INFOCOM, 2005.
- quality through path switching. In *INFOCOM*, 2005.
 [52] L. Wang, K. Park, R. Pang, V. S. Pai, and L. L. Peterson.
 Reliability and security in the CoDeeN content distribution network. In *USENIX*, 2004.
- [53] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *HotNets-II*, 2003.
- [54] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A lightweight network location service without virtual. In SIGCOMM, 2005.
- [55] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In OSDI, 2004.
- [56] H. Zheng, E. K. Lua, M. Pias, and T. Griffin. Internet routing policies and round-trip-times. In *PAM*, 2005.

Fidelity and Yield in a Volcano Monitoring Sensor Network

Geoff Werner-Allen*, Konrad Lorincz*, Jeff Johnson†, Jonathan Lees‡, and Matt Welsh*

* Division of Engineering and Applied Sciences, Harvard University

† Dept. of Earth Sciences, University of New Hampshire

‡ Dept. of Geological Sciences, University of North Carolina

Abstract

We present a science-centric evaluation of a 19-day sensor network deployment at Reventador, an active volcano in Ecuador. Each of the 16 sensors continuously sampled seismic and acoustic data at 100 Hz. Nodes used an event-detection algorithm to trigger on interesting volcanic activity and initiate reliable data transfer to the base station. During the deployment, the network recorded 229 earthquakes, eruptions, and other seismoacoustic events.

The science requirements of reliable data collection, accurate event detection, and high timing precision drive sensor networks in new directions for geophysical monitoring. The main contribution of this paper is an evaluation of the sensor network as a scientific instrument, holding it to the standards of existing instrumentation in terms of data *fidelity* (the quality and accuracy of the recorded signals) and *yield* (the quantity of the captured data). We describe an approach to *time rectification* of the acquired signals that can recover accurate timing despite failures of the underlying time synchronization protocol. In addition, we perform a detailed study of the sensor network's data using a direct comparison to a standalone data logger, as well as an investigation of seismic and acoustic wave arrival times across the network.

1 Introduction

Sensor networks are making inroads into a number of scientific explorations, including environmental monitoring [1, 25], habitat monitoring [2, 10, 23], and structural monitoring [16, 14, 30]. In each of these domains, the use of low-power wireless sensors offers the potential to collect data at spatial and temporal scales that are not feasible with existing instrumentation. Despite increased interest in this area, little has been done to evaluate the ability of sensor networks to provide meaningful data to domain scientists. A number of challenges confound such an effort, including node failure, message loss, sensor calibration, and inaccurate time synchronization. To successfully aid scientific studies, sensor networks must be held to the high standards of current scientific instrumentation.

In this paper, we take a hard look at the performance of a wireless sensor network deployed on an active volcano. We evaluate its effectiveness as a scientific instrument using two metrics: *data fidelity* and *yield*. Data fidelity encompasses the quality and consistency of retrieved seismoacoustic signals, while data yield reflects the quantity of data delivered by the network.

Typical volcano monitoring studies employ GPS-synchronized data loggers recording both seismic and acoustic signals. These provide high data fidelity and yield but are bulky, power hungry, and difficult to deploy. Existing analog and digital telemetry is similarly cumbersome. The use of wireless sensors could enable studies involving many more sensors distributed over a larger area. However, the science requirements pose a number of difficult challenges for sensor networks. First, seismoacoustic monitoring requires high data rates, with each node sampling multiple channels at 100 Hz. Second, signal analysis requires complete data, necessitating reliable data collection. Third, volcano studies compare signals across multiple sensors, requiring that collected data be accurately timestamped against a GPS-based global clock.

The core contribution of this paper is an analysis of the efficacy and accuracy of a volcano-monitoring sensor network as a scientific instrument. This is the first paper to our knowledge to take a science-centric view of a sensor network with such demanding data-quality requirements. In this paper, we evaluate the data collected from a 19-day field deployment of 16 wireless sensors on Reventador volcano, Ecuador, along the following axes:

- Robustness: We find that the sensor nodes themselves
 were extremely reliable but that overall robustness was
 limited by power outages at the base station and a single
 three-day software failure. Discounting the power outages and this single failure, mean node uptime exceeded
 96%.
- Event detection accuracy: Our network was designed to trigger data collection following volcanic events such as earthquakes and eruptions. We measure the accuracy of our distributed event-detection algorithm, finding that the algorithm has a zero false positive rate. However,

the network failed to detect many seismic events due to a poor choice of event-detection parameters and limitations of our data collection protocol.

- Data transfer performance: We evaluate the ability of our data collection protocol to transfer complete signals following an event. We find a 90th percentile event yield (fraction of nodes for which all data for an event was collected) of 94% and a latency of 63 sec per radio hop for downloading 60 sec worth of data.
- Timing accuracy: Data collected by each node must be timestamped to within a single sample time (10 ms) to enable seismological analysis. We evaluate the stability of the underlying time synchronization protocol (FTSP [11]), and develop a novel approach to time rectification that accurately timestamps each sample despite failures of the FTSP protocol. We show that this approach recovers timing with a 90th-percentile error of 6.8 msec in a 6-hop network.
- Data fidelity: Finally, we take a seismological view of the captured data and present a head-to-head comparison of data recorded by our sensor network against a colocated data logger. We also evaluate the consistency of the recorded signals in terms of seismic and acoustic wave arrival times across the network, showing that the data is consistent with expected physical models of the volcano's activity.

The rest of this paper is organized as follows. The next section provides background on the use of wireless sensors for volcano monitoring and outlines the underlying science goals. In Section 3 we describe the architecture of our system and the field deployment at Reventador. Sections 4 through 8 present a detailed analysis of the network's performance along each of the evaluation metrics described above. Section 9 discusses related work and Section 10 presents several lessons learned from the deployment. Section 11 outlines future work and concludes.

2 Background

Scientists monitor volcanoes for two non-exclusive reasons: (1) to monitor hazards by assessing the level of volcanic unrest; and (2) to understand physical processes occurring within the volcano, such as magma migration and eruption mechanisms [21, 12]. The most common instrument used is the seismometer, which measures ground-propagating elastic radiation from both sources internal to the volcano (e.g., fracture induced by pressurization) and on the surface (e.g., expansion of gases during an eruption) [12]. In addition, microphones are sometimes employed to record *infrasound*, low-frequency (< 20 Hz) acoustic waves generated during explosive events. Infra-

sound is useful for differentiating shallow and surface seismicity and for quantifying eruptive styles and intensity [7].

2.1 Existing volcano instrumentation

The type of instrumentation used to study volcanoes depends on the the science goals of the deployment. We are focused on the use of wireless sensors for temporary field deployments involving dozens of sensor stations deployed around an expected earthquake source region, with internode spacing of hundreds of meters. A typical campaign-style deployment will last weeks to months depending on the activity level of the volcano, weather conditions, and science requirements.

Geophysicists often use standalone dataloggers (e.g., Reftek 130 [20]) that record signals from seismometers and microphones to a flash drive. These data loggers are large and power-hungry, typically powered by car batteries charged by solar panels. The sheer size and weight precludes deployments of more than a small number of stations in remote or hazardous areas. Additionally, data must be retrieved manually from each station every few weeks, involving significant effort. Analog and digital radio telemetry enables real-time transmission of data back to an observatory. However, existing telemetry equipment is very bulky and its limited radio bandwidth is a problem for collecting continuous data from multiple channels.

2.2 Sensor network challenges

Wireless sensor networks have the potential to greatly enhance understanding of volcanic processes by permitting large deployments of sensors in remote areas. Our group is one of the first to explore the use of wireless sensor networks for volcano monitoring. We have deployed two wireless arrays on volcanoes in Ecuador: at Volcán Tungurahua in July 2004 [27], and at Reventador in August 2005 [28]. The science requirements give rise to a number of unique challenges for sensor networks, which we outline below.

High-resolution signal collection: Data from seismometers and microphones must be recorded at relatively high data rates with adequate per-sample resolution. A sampling rate of 100 Hz and resolution of 24 bits is typical. This is in contrast to sensor networks targeting low-rate data collection, such as environmental monitoring [23, 25].

Triggered data acquisition: Due to limited radio bandwidth (less than 100 Kbps when accounting for MAC overhead), it is infeasible to continuously transmit the full-resolution signal. Instead, we rely on triggered data collection that downloads data from each sensor following a significant earthquake or eruption. This requires sensor nodes to continuously sample data and detect events of interest. Event reports from multiple nodes must be collated to accurately detect *global* triggers across the network.

Timing accuracy: To facilitate comparisons of signals across nodes, signals must be timestamped with an accu-

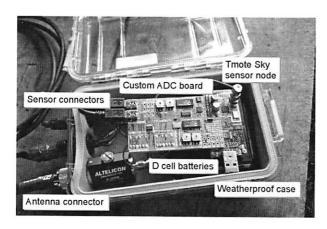


Figure 1: Our wireless volcano monitoring sensor node.

racy of one sample time (i.e., 10 ms at 100 Hz). Data loggers generally incorporate a GPS receiver and use low-drift oscillators to maintain accurate timing. However, equipping each sensor node with a GPS receiver would greatly increase power consumption and cost. Instead, we rely on a network time synchronization protocol [4, 11] and a *single* GPS receiver. However, correcting for errors in the time synchronization protocol requires extensive post-processing of the raw timestamps.

3 System Architecture

In this section we provide a brief overview of the design of our volcano monitoring sensor network and details of the deployment at Reventador. In an earlier magazine article [28] we describe the system and deployment in more detail, although we have not previously published results evaluating its performance.

3.1 Sensor hardware

Our wireless sensor node (Figure 1) is based on the TMote Sky [13] platform, which integrates a TI MSP430 processor, 10 KB of SRAM, 48 KB of program ROM, 1 MByte of flash memory, and a Chipcon CC2420 radio. All software is implemented in TinyOS [5]. We designed a custom sampling board that provides four channels of 24-bit analog-to-digital conversion (TI AD7710).

Nodes were interfaced to either a single-axis seismometer (GeoSpace GS-11) or three seismometers in a triaxial configuration (GeoSpace GS-1). Both sensors are passive instruments; ground motion generates a voltage which is amplified and digitized by the sampling board. In addition, each node was attached to an omnidirectional microphone (Panasonic WM-034BY). This microphone has been used in other infrasonic monitoring studies [7].

Each node was equipped with an 8.5 dBi omnidirectional antenna mounted on 1.5 m of PVC pipe. This permitted line-of-sight radio range of over 1 km without amplification; nodes were typically placed 200-400 m apart in

our deployment. Nodes were powered by two D-cell batteries with a lifetime of approximately 1 week. Each node was enclosed in a weatherproof Pelican case.

Several other pieces of hardware complete the system. FreeWave radio modems provided a long-distance radio link between the sensor array and the volcano observatory, 4.6 km away. A laptop located at the observatory logged data and was used to monitor and control the network. Finally, to establish a global timebase, we used a single Crossbow MicaZ [3] mote interfaced to a GPS receiver (Garmin OEM 18 LVC). The GPS receiver provided a 1 Hz pulse that is accurate to GPS time within 1 μ s, and acted as the root of the network time synchronization protocol as described in Section 7.

3.2 Network topology and status monitoring

Nodes form a multihop routing tree rooted at the gateway node that is physically attached to the FreeWave modem; we use a variant of MintRoute [29] that uses the CC2420's Link Quality Indicator metric to select routing paths. Each node transmits a *status message* every 10 sec that includes its position in the routing tree, buffer status, local and global timestamps, battery voltage, and other information. In addition, the base station can issue a *command* to each node, instructing it to respond with an immediate status message, start or stop data sampling, and set various software parameters. Commands are propagated using a simple flooding protocol. The *Deluge* protocol [6] was also used to permit over-the-air reprogramming and rebooting of nodes.

3.3 Event detection and data collection

Because of the high data rates involved (600-1200 bytes/sec from each node) it is infeasible to continuously transmit all sensor data. Rather, nodes are programmed to locally detect interesting seismic events and transmit event reports to the base station. If enough nodes trigger in a short time interval, the base station attempts to download the last 60 sec of data from each node. This design forgoes continuous data collection for increased resolution following significant seismic events, which include earthquakes, eruptions, or long-period (LP) events, such as tremor. The download window of 60 sec was chosen to capture the bulk of the eruptive and earthquake events, although many LP events can exceed this window (sometimes lasting minutes or hours). To validate our network against existing scientific instrumentation, our network was designed for high-resolution signal collection rather than extensive in-network processing.

During normal operation, each node continuously samples its seismic and acoustic sensors at 100 Hz, storing the data to flash memory. Data is stored as 256-byte *blocks* in the flash. Each block is tagged with the *local timestamp* corresponding to the first sample in the block. This

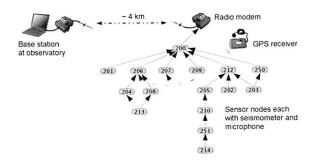


Figure 2: Sensor network architecture. Nodes form a multihop routing topology, relaying data via a long-distance radio modem to the observatory. A GPS receiver is used to establish a global timebase. The network topology shown here was used during our deployment at Reventador.

timestamp is later mapped onto a global time reference as described in Section 7. The 1 Mbyte flash is treated as a circular buffer storing approximately 20 min of data.

In addition, nodes run an event detection algorithm that computes two exponentially-weighted moving averages (EWMA) over the input signal with different gain settings. When the ratio between the two EWMAs exceeds a threshold, the node transmits an event report to the base station. If the base station receives triggers from 30% of the active nodes within a 10 sec window, it considers the event to be well-correlated and initiates data collection.

Our reliable bulk-transfer protocol, called *Fetch*, operates as follows. The base station waits for 30 sec following an event before iterating through all nodes in the network. The base sends each node a command to temporarily stop sampling, ensuring the event will not be overwritten by subsequent samples. For each of the 206 blocks in the 60 sec window, the base sends a *block request* to the node. The node reads the requested block from flash and transmits the data as a series of 8 packets. After a short timeout the base will issue a repair request to fill in any missing packets from the block. Once all blocks have been received or a timeout occurs, the base station sends the node a command to resume sampling and proceeds to download data from the next node.

3.4 Deployment on Volcán Reventador

Our deployment at Reventador took place between August 1–19, 2005. Reventador is an active volcano located in northern Ecuador, about 100 km from Quito. During this time, Reventador's activity consisted of small explosive events that ejected ash and incandescent blocks several times a day. Associated seismicity included numerous explosion earthquakes as well as extended-duration shaking (tremor) and shallow rock-fracturing earthquakes.

We deployed 16 sensor nodes on the upper flanks of Reventador, as shown in Figure 3, over a 3 km linear configuration radiating away from the vent. The resulting multihop topology is shown in Figure 2. The upper flanks of

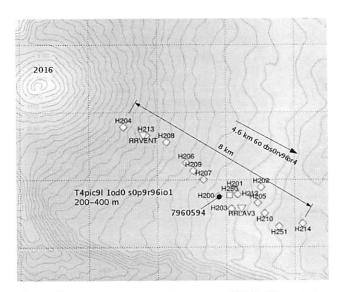


Figure 3: Map of sensor deployment at Volcán Reventador. In addition to the 16 sensor nodes, two broadband seismometers with data loggers (RRVENT and RRLAV3) were colocated with the network.

the volcano were completely deforested by a large eruption in November 2002, allowing for line-of-sight radio communication between adjacent sensor nodes. Two standalone seismic stations, consisting of a broadband sensor, a Reftek 130 data logger with 1 GByte flash memory cards, and a GPS receiver for timestamping, were colocated with sensor nodes. The data from these stations is essential to our network validation, described in Sections 5 and 7. The base station was located at a small hotel 4.6 km from the deployment site. The sensors were deployed for a total of 19 days, during which time the network recorded data from 229 earthquakes, eruptions, and tremor events, logging 107 MBytes of data. The long hike and lack of roads prevented frequent returns to the deployment site, although we returned several times to change batteries and perform other network maintenance.

4 Network Robustness

The first evaluation metric that we consider is the *robust-ness* of the sensor network. Sensor network deployments have typically been plagued by failures of individual nodes and the support infrastructure. Clearly, robustness has a direct effect on the resulting data yield. Our evaluation shows that while nodes exhibited very high uptimes, the base station infrastructure was very unreliable, and a single bug affecting the Deluge protocol caused a three-day outage of the entire network.

4.1 Overall network uptime

Figure 4 shows the number of nodes reporting over each 10-minute interval during the entire 19-day deployment. A node is included in the count if any of its status messages

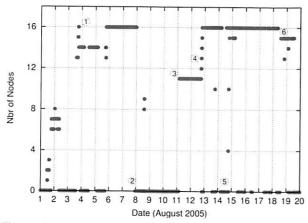


Figure 4: Nodes reporting over time. This figure shows the number of nodes reporting over each 10 min window during the 19-day deployment period. The annotations (1) through (6) are described in the text.

were received at the base station during the 10-minute window. Annotations show several significant events that occurred during the deployment. The network was installed in two phases of 8 nodes each, the first on August 1 and the second on August 3. At label (1) the entire 16 node network is operational. However, initial software misconfiguration required rebooting several nodes during a third visit to the deployment site on August 5. The network then ran with 16 nodes active for a little more than 2 days.

At label (2) on August 8, a software command was transmitted to reboot the network, using Deluge [6], in an attempt to correct the time synchronization fault described in Section 7. This caused a software failure affecting all nodes, with only a few reports being received at the base station later on August 8. After repeated attempts to recover the network, we returned to the deployment site on August 11 (label (3)) to manually reprogram each node. However, only 11 nodes could be reached before nightfall, forcing a return to the observatory. On August 12 (label (4)) we returned to the deployment site and reprogrammed the remaining 5 nodes.

From August 13 through 18, all 16 nodes were reporting nearly continuously. The intermittent failures (label (5)) were caused by power outages at the observatory, causing the base station laptop and radio modem to fail. During these times no data was logged by the base station although the sensor nodes themselves were probably operational, since all nodes would report when the base station recovered.

Several days before the end of the deployment, node 204, located closest to the vent, stopped reporting data (label (6)). When the network was disassembled we discovered that the antenna mast had been destroyed, most likely by a bomb ejected from the volcano during an eruption, although the node itself remained intact. This failure underscores the importance of remote telemetry for acquiring

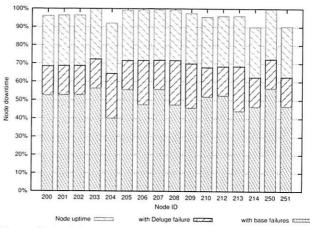


Figure 5: Individual node uptimes. This figure shows the percentage of time that each node reported status messages during the 19-day deployment. Shown separately are the apparent node uptimes caused by the whole-network outage and base station outages. While the former was true sensor node failure, the latter did not seem to affect the sensor nodes themselves.

data at hazardous volcanoes.

4.2 Individual node uptime

Figure 5 shows the uptime for each node during the 19-day deployment. Each bar consists of three portions. The lowest portion is the *apparent* uptime of each node accounting for both the base station failures and single 3-day software outage. Because base station failures did not affect individual nodes, the middle bar shows the apparent uptime including only the 3-day outage. In this case, the mean node uptime is 69%. However, with the 3-day outage factored out, nodes achieved an average uptime of 96%. These numbers are encouraging and suggest that the sensor nodes were very reliable in spite of the software crash.

Based on discussions with the authors of Deluge, we believe this failure was caused by a single bug in the InternalFlash TinyOS component (which has since been fixed). This bug prevented Deluge from storing critical state information, causing nodes to reboot continuously at short intervals. We did not see this behavior in the lab before deployment, although we had not rigorously tested this portion of the code. In retrospect, it was optimistic of us to rely on a complex network reboot protocol that had not been field-tested. Deluge was removed from the binary used during the network reprogram following the failure; it was replaced with a simpler mechanism to reboot individual nodes using a radio command.

4.3 Discussion

Failures of the base station infrastructure were a significant source of network downtime during the deployment. This contrasts with common assumptions that the base station is generally reliable and operating on a continuous power source. This was our expectation prior to the deployment, and we did not make adequate preparations for the intermit-

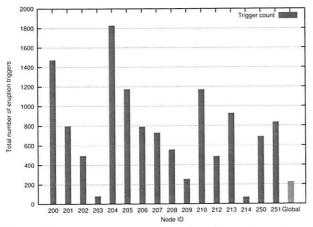


Figure 6: Event triggers per node. This figure shows the total number of event triggers reported by each node. It demonstrates a wide variation in trigger rates that cannot be attributed only to varying node uptimes. For example, node 204 had the lowest uptime but the largest number of event triggers.

tent electrical supply at the observatory. A backup diesel generator was used during nightly power outages, with extra laptop and car batteries supplying power when it failed. However, this approach was not ultimately successful.

It may be surprising that node uptime is not related to depth in the routing tree. This suggests that if a node is "down" (i.e., we do not receive any status messages from it during a 10-minute window) that it is still active and routing packets for its children in the tree, even as its own status messages are being lost. An alternate explanation is that a node could select an alternate parent in the routing topology when its parent fails. However, our analysis of the routing topology (Figure 2) does not support this view, since nodes rarely use more than one parent. For example, node 214 *always* routes data through node 251. The volcano-induced failure of node 204 near the end of the deployment is the only notable failure of a single node.

5 Event Detector Accuracy

Our network was designed to capture interesting volcanic signals. Thus, it is critical that the system correctly identify and report such events. This section evaluates our event detection algorithm both in terms of the number and rate of event triggers as well as its ability to detect scientifically interesting events.

5.1 Event triggers per node

Figure 6 shows the total number of events reported by each node during the deployment. It shows a wide variation in the event trigger rate, from 70 triggers for node 213 to 1830 triggers for node 204. Variation in the trigger rate can be attributed to many factors, including the location of the node, the orientation of the seismometer, and the quality of the seismometer-to-ground coupling. Note that the trigger rate does not seem to be related to distance from the vent.

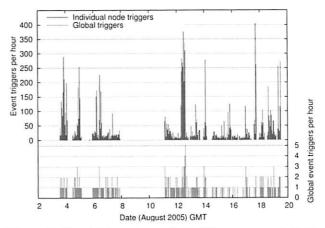


Figure 7: Event triggers over time. The upper graph shows the total number of individual node triggers per hour. The lower graph shows the corresponding number of global triggers. Reventador's varying seismic activity generated between 0 to 5 global triggers per hour.

Although node 204 was closest to the vent and reported the most triggers, nodes 200, 205, and 210 all had high trigger counts despite being significantly farther away.

5.2 Event triggers over time

Figure 7 shows both the number of individual node and global event triggers over each hour. We observe that the volcano's activity varied greatly, generating trigger counts ranging between 2 and 405 events per hour when the network was online. This activity translates into up to 5 global event triggers an hour, each initiating a Fetch download cycle of the associated data.

The volcano's bursty and unpredictable activity makes the network's design more challenging than systems designed for statically-scheduled data collection. The data collection protocol, based on our earlier deployment at Tungurahua [27], assumed that events would be rare and that it would be unnecessary to simultaneously record signals for one event while downloading another. As a result, we missed a number of impressive back-to-back eruptions typical of the activity at Reventador. It is worth noting that the variable number of event reports is itself a measure of the volcano's activity level and could be used to assess hazard levels.

5.3 Event detector accuracy

The network detected 229 eruptions, explosions, earth-quakes, and tremor events during the deployment. Ideally, we would like to assess its accuracy in terms of the fraction of true events detected, as well as the false positive rate. Given the high degree of coherence required by the global event detector (requiring 30% of the active nodes to trigger within a short time window), we would be surprised if the sensor network recorded any false events. Indeed, all of the signals we did capture appear to be based on true volcanic

activity, indicating a zero false positive rate.

We intended to apply our event detection algorithm to the signals collected by the two broadband seismic stations to establish the algorithm's accuracy. Unfortunately, we found this to be difficult for several reasons. First, each of the broadband stations suffered intermittent power and software failures, either preventing them from logging any data, or corrupting the collected signals or timestamps. Thus, even in those cases where broadband data is available, it is not always accurate. Second, the broadband stations deployed a more sensitive seismometer with a much wider frequency response. The geophones used by our sensor nodes have a corner frequency of 4.5 Hz, while the broadband sensors have a corner frequency of 0.033 Hz. Additionally, the broadband seismometers are much more sensitive, generating voltages of 800 V/m/sec, whereas the geophones have a sensitivity of only 32 V/m/sec. As a result, the broadband sensors are able to detect much weaker seismic signals.

We focus our attention on a single day of data where the broadband stations were recording clean data and the sensor network was relatively stable. One of the authors, a seismologist, visually extracted events from the broadband data; during this 24-hour period, a total of 589 events were recorded by the broadband sensors. During the same time, the sensor network triggered on just 7 events, suggesting that our detection accuracy is very low (about 1%).

The network could have failed to detect a seismic event for one of four reasons: (1) failure of individual nodes; (2) failure of the base station or radio modem; (3) the low sensitivity of our seismometers; or (4) failure of the event detection algorithm itself. To factor out the increased sensitivity of the broadband seismometers, we only consider the 174 events with SNR ≥ 10 from both stations, which we expect the geophones should have been able to detect as well. Also, Section 4 has already addressed the question of uptime, so we focus here on the inherent accuracy of the event detector when the network was operating correctly. 136 of the 174 broadband events occurred during times when the network was operational. Taking these two factors into account, the network's detection accuracy is still only about 5%.

Recall that during a Fetch download cycle, nodes disabled sampling to avoid overwriting data in flash. Download cycles could take up to several minutes *per node* (see Section 6), meaning that there are significant time windows when the network was unable to detect new events. During the Fetch cycles on August 15, the broadband stations recorded 42 events, 24% of the total events detected. This indicates that, all else being equal, the sensor network could have detected approximately 24% more events had we designed the protocol to sample and download simultaneously. We plan to add this feature in the next version of our system.

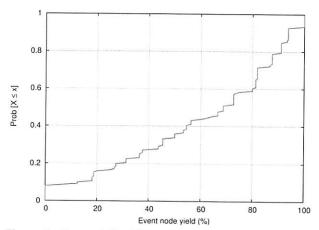


Figure 8: Event yield. This graph shows a CDF of the event yield for each of the 229 events recorded during the entire deployment. Event yield is the fraction of active nodes from which a complete 60 sec signal was downloaded following an event.

In the end, we believe that our low detection rate is the result of the parameters used in the EWMA-based event detection algorithm. These parameters were chosen prior to the deployment, based on our experience with detecting infrasonic events at a different volcano [27]. We did not experiment with modifying them in the field. Indeed, using our algorithm with these same parameters on the broadband data for August 15 detects only 101 events, a fraction of the events chosen manually by an expert. We plan to tune our event-detection parameters for future deployments based on the data collected by the broadband stations.

6 Data Collection Performance

In this section we assess the performance of the *Fetch* data collection protocol. We evaluate Fetch in terms of its *yield*, its ability to successfully collect requested data; and its *latency*, the time to download events from the network.

6.1 Data yield

We define the *event yield* of a Fetch transfer as the fraction of nodes for which the entire 60 sec signal was successfully downloaded following an event. The calculation only considers those nodes that were active at the time of the event detection (Figure 4). For example, if 10 nodes were active during an event, then the event yield is defined in terms of 10 nodes. Note that the Fetch protocol attempts to download a signal from all active nodes, even those that did not detect the event.

Figure 8 shows a CDF of the event yield for all 229 events recorded during the deployment. As the figure shows, the median event yield was 68.5% and the 90th percentile was 94%. The yield can be affected by several factors. First, the protocol will abort a transfer from a node after re-requesting the same block more than 20 times, or if the transfer from a single node exceeds 10 minutes. Sec-

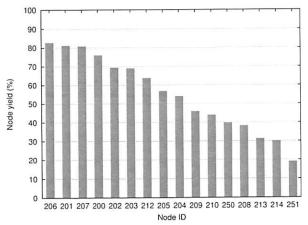


Figure 9: Node yield. This graph shows the node yield for each of the 16 nodes over the entire deployment, defined as the probability that an event was successfully downloaded from a node, as long as that node is active during the corresponding event detection.

ond, because sampling is disabled while performing a data transfer, if two back-to-back events occur a node may not end up storing data for the second event.

Next, we look at the *node yield* which we define as the probability that an event was successfully downloaded from a given node. Like the event yield, the calculation only considers those nodes that were active at the time of each event detection. Node yield can be affected by several factors. The depth and radio link quality of a node's routing path to the base station affect packet loss rate and thereby the likelihood of a Fetch timeout. Additionally, two nodes outfitted with triaxial seismometers (nodes 250 and 251) sample and store twice as much data as the others, increasing the probability of a timeout. Finally, a bug in our control application caused node 250 to sample data continuously, even during a Fetch operation. As a result, this node was more likely to overwrite an event stored in flash before it could be downloaded.

Figure 9 shows the node yield for each of the nodes. We can see how the factors mentioned above affected performance. First, the nodes with the highest yield (above 80%) tend to be within two hops from the root (see Figure 2). However, despite being within two or three hops, node 209 had a fairly low yield. This is explained by the fact that node 209 had a poor link to its closest parent, node 200. In fact, although most nodes had a stable parent throughout the deployment, node 209 used node 200 as its parent only 33% of the time and nodes 206 and 207 the remaining 66% of the time. Node 213 also switched parents between nodes 204 and 208, but unlike node 209 it was always three hops away. Node 214 was the farthest node in terms of hopcount and as a result had one of the lowest yields. The larger amount of data was also a factor for the four-channel nodes, 250 and 251. In addition, node 251 was five radio hops from the gateway.

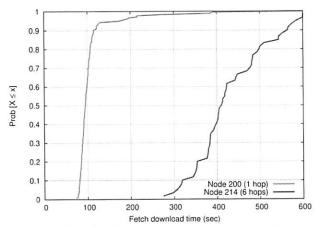


Figure 10: **Distribution of Fetch latency for two nodes.** The latency for a Fetch download depends on the depth of the node in the routing tree, which affects both command propagation latency and reliability of the routing path. Node 200 is located 1 hop from the sink and node 214 is located 6 hops away.

6.2 Fetch latency

Transfer latency directly impacts data yield. Because we disabled sampling on each node (apart from node 250) during a Fetch download cycle, the duration of the data transfer also affects a node's ability to record back-to-back events.

The median latency for Fetch operations (downloading 60 sec worth of data from a single node) was 186 sec and the 90th percentile was 444 sec. Unsurprisingly, latency varies with the depth of the node in the routing tree. Figure 10 compares Fetch latency for nodes 200 and 214, located 1 and 6 hops away from the sink, respectively. Node 200 had a median Fetch latency of 94 sec, while node 214 had a median latency of 409 sec, about 63 sec per hop. This is due to both increased delay for propagating Fetch command messages, as well as increasing packet loss and retransmission overheads as the data flows over multiple hops to the base.

Fetch was initially designed to support reliable downloads of infrequent events and we did not anticipate the need to capture back-to-back signals. Unfortunately, these were common at Reventador, and may necessitate a redesign. For example, it may be possible to reduce latency by streaming multiple blocks in one request and reconstructing partial blocks after a burst. Caching recentlyreceived blocks on intermediate nodes could reduce latency for repair requests [14]. However, such changes would greatly increase the complexity of the protocol. For this deployment we opted to prioritize simplicity and stability over performance.

7 Time Rectification and Accuracy

When analyzing seismoacoustic data acquired at volcanoes, accurate timing of recorded signals is paramount. Studying volcanic source processes necessitates precisely identifying the arrival time of P- and S-waves at each sensor. Also, correlating signals across the sensor array requires accurately timestamping each sample. Ideally, timing should be accurate to within one sample interval, or 10 ms when sampling at 100 Hz. As described earlier, we opted to use a single GPS receiver and employ a multihop time-synchronization protocol to establish a global time-base. The protocol worked well in laboratory experiments. However, it experienced significant failures in the field, requiring extensive postprocessing of the data to recover accurate timing for each signal.

In this section, we provide an overview of the time synchronization errors observed in the field. We then present a novel *time rectification* technique that allows us to recover accurate timing despite protocol failures. We evaluate our approach through lab experiments with a known, ground-truth timebase, and by comparing our signals with signals recorded by the colocated data loggers. This paper is the first to our knowledge to evaluate the stability of a multi-hop time synchronization protocol during a lengthy sensor network field deployment.

7.1 Time synchronization architecture

We chose to use the Flooding Time Synchronization Protocol (FTSP) [11], an existing protocol developed for wireless sensor nodes. In the original FTSP work [11], timing errors of less than 67 μ sec were reported for an 11-hop network of Mica2 nodes. We verified in our testbed that FTSP provided a 90th-percentile time error of under 2.1 ms in a 5-hop linear network of TMote Sky nodes.

A single MicaZ sensor node was used as the root of the FTSP synchronization tree. It interfaced to a Garmin GPS receiver and received a 1 Hz interrupt synchronized to within 1 µsec of the GPS "pulse per second" signal. When the interrupt is raised, the node records the GPS time and corresponding FTSP global time and sends a short message containing this information to the base station. Each sensor node runs the FTSP protocol which maintains a global timebase. Every 10 sec, each node records its local time and the corresponding FTSP global time, sending this information in its status message to the base station. Finally, as each node records data, the first sample of each block is marked with the node's local time. After downloading data from each node following an event, this local time can be used to recover the time for each sample in the block.

Therefore, we have three relevant timebases: the *local time* at each node; the *global time* established by the FTSP protocol; and the *GPS time* recorded by the FTSP root. The information in the nodes' status messages can be used to map local time to global time, and the information in the GPS node's status messages can be used to map global time to GPS-based GMT.

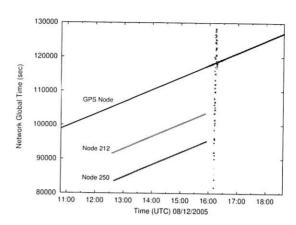


Figure 11: Example of FTSP instability observed during field deployment: The global time value reported by sensor nodes and the GPS node is plotted against the time that the base station received the corresponding status messages. All nodes are initially synchronized, but starting at 1230 GMT, nodes 212 and 250 report incorrect global times for the next 4.5 hours. When the nodes eventually resynchronize, the global timestamps of other nodes initially experience some instability.

7.2 FTSP failures in the field

In the absence of failures, this mapping would be a straightforward process. However, in the field, we noticed that
nodes would occasionally lose synchronization with the
rest of the network and report FTSP global times with significant errors, sometimes exceeding several hours. We
suspect that the sparse deployment conditions at the volcano might have led to different behavior in the time synchronization protocol than in the lab. For example, occasional message loss or failure of a neighbor could cause
the node's global time to drift from the rest of the network.
However, in lab tests that constrained the network topology
we did not observe these instabilities.

Figure 11 shows an example of the FTSP instability observed in the field. The global time reported by two nodes suddenly jumps off by several hours, and the nodes do not resynchronize until rebooted 4.5 hours later. It turns out that two bugs conflated to cause this problem. First, it was discovered that the TinyOS clock driver would occasionally return bogus local timestamps. This bug was fixed in February 2006, several months after our deployment. Second, FTSP does not check the validity of synchronization messages, so a node reading an incorrect value for its local clock can corrupt the state of other nodes, throwing off the global time calculation. To our knowledge, few if any sensor network deployments have attempted to use network time synchronization protocols for extended periods. In addition, ours may have been the first deployment of FTSP on the TMote Sky platform where the clock driver bug manifested itself.

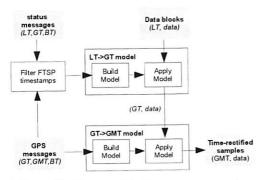


Figure 12: Time rectification process overview.

The failures of the time synchronization protocol make establishing the correct GPS-based timestamp for each data sample extremely challenging. Our *time rectification* approach filters and remaps recorded timestamps to accurately recover timing despite these failures. The time rectification process is illustrated in Figure 12. The first step is to *filter* the global timestamps recorded by each node, discarding bogus data. Second, we build a model mapping the local time on each node to FTSP-based global time. Third, we use the GPS timestamp information to build a second model mapping FTSP time to GMT. Finally, both models are applied to the timestamps recorded in each data block producing a GMT time for each sample.

7.3 Timestamp filtering

We begin by filtering out status messages appearing to contain incorrect global timestamps. To do this, we correlate global timestamps from each node against a common reference timebase and reject those that differ by more than some threshold. For this, we use the base station laptop's local time, which is *only* used for filtering FTSP timestamps, not for establishing the correct timing. The filtering process in is many ways similar to prior work [17, 18] on detecting adjustments in network-synchronized clocks.

We use the following abbreviations: LT is the local time of a node; GT is the FTSP global time; BT is the base station's local time; and GMT is the true GMT from the GPS signal. Each GPS status message logged by the base station consists of the triple (GT, GMT, BT). We use linear regression on this data to produce a reference timebase mapping BT to GT. For each node status message logged by the laptop (LT, GT, BT), we map BT to the expected GT_{ref} using the reference timebase. If $|GT_{ref} - GT| > \delta$, we discard the status message from further consideration. We use a threshold of $\delta = 1$ sec. Although radio message propagation and delays on the base station can affect the BT for each status message, a small rejection threshold δ makes it

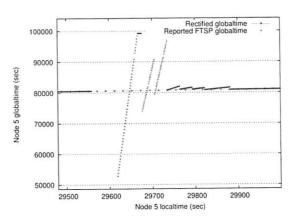


Figure 13: Time rectification example. The raw (LT, GT) pairs collected from the node show that it experiences a period of FTSP instability. The time rectification process removes the errant timestamps creating an accurate mapping between LT and GT created using a linear regression on the remaining timestamps.

unlikely that any truly incorrect FTSP timestamps pass the filter. Indeed, of the 7.8% of timestamps filtered out, the median *GT* error was 8.1 hours.

7.4 Timestamp rectification

The goal of *time rectification* is to assign a GMT timestamp to each sample in the recorded data. In order to do so, we build two models: one mapping a node's local time to global time, and another mapping global time to GMT. From those status messages that pass the filter, we build a piecewise linear model mapping *LT* to *GT* using a series of linear regressions. Models are constructed for each node separately, since local times vary significantly between nodes. Each regression spans up to 5 minutes of data and we initiate a new regression if the gap between subsequent (*LT*, *GT*) pairs exceeds 5 minutes. Each interval must contain at least two valid status messages to construct the model. We take the *LT* value stored in each data block and use this model to recover the corresponding *GT* value.

The next step is to map global time to GMT. Each of the GPS node's status messages contain a (GT, GMT) pair. As above, we build a piecewise linear model mapping GT to GMT, and apply this model to the GT values for each data block. Finally, we assign a GMT value to each sample contained in the block, using linear interpolation between the GMT values assigned to the first sample in each block. This process makes no assumptions about sampling rate, which varies slightly from node to node due to clock drift.

7.5 Evaluation

Evaluating this time rectification process has proved difficult, primarily because we have no ground truth for the timing of the signals recorded in the field. However, by re-

¹We assume that the global time reported by the GPS node is always correct; indeed, the definition of "global time" is the FTSP time reported by the GPS node. We verified that the FTSP instability affecting the sensor nodes did not occur on the GPS node, most likely because the MicaZ uses a different processor that is unaffected by the clock driver bug.

	Raw error	Rectified error
1 hop, 50th percentile	1.52 ms	1.42 ms
1 hop, 90th percentile	9.86 ms	6.77 ms
6 hops , 50th percentile	2.63 ms	2.18 ms
6 hops, 90th percentile	13.5 ms	6.8 ms

Figure 14: Timestamp errors in a 6-hop lab testbed: This table shows the 50th and 90th-percentile timing errors on both the raw FTSP timestamps, and rectified timestamps.

producing the deployment conditions in the lab, we have been able to measure the accuracy of the recovered timing in a controlled setting. In addition, as described earlier, two GPS-synchronized data loggers were colocated with our sensor network, providing us the opportunity to directly compare our time-rectified signals with those recorded by conventional instrumentation.

Our first validation took place in the lab. Feeding the output of a signal generator to both a miniature version of our sensor network and to a Reftek 130 data logger allowed us to directly compare the data between both systems. The miniature network consisted of a single sensor node, routing gateway, and GPS receiver node. The same software was used as in the field deployment. The Reftek 130 logs data to a flash memory card and timestamps each sample using its own GPS receiver.

The results showed a consistent 15 ms offset between the time-rectified signals recorded by the sensor node and the Reftek data logger. We discovered that this offset was due to delays introduced by the digital filtering performed by the ADC on our sensor board (see Section 3.1). Adjusting for this delay resulted in an indiscernible offset between the sensor node and Reftek signals. While this experiment does not reproduce the full complexity of our deployed network, it does serve as a baseline for validation.

In the second lab experiment, we set up a network of 7 sensor nodes in a 6-hop linear topology. The topology is enforced by software, but all nodes are within radio range of each other, making it possible to stimulate all nodes simultaneously with a radio message. Each node samples data and sends status messages using the same software as the field deployment. The FTSP root node periodically transmits a beacon message. On reception of the beacon, each node records the FTSP global timestamp of the message reception time (note that reception of the beacon message is not limited by the software-induced topology). Because we expect all nodes to receive this message at the same instant, modulo interrupt latency jitter, we expect the FTSP time recorded by each node to be nearly identical. The FTSP root also records the time that the beacon was transmitted, accounting for MAC delay. The experiment ran for 34 hours, during which time FTSP experienced instabilities similar to those seen during our deployment.

This allows us to compare the *true* global time of each beacon message transmission and the *apparent* global time on each receiving node, both before and after subjecting

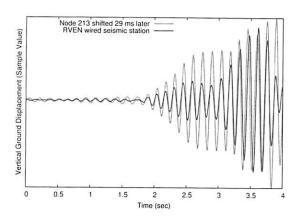


Figure 15: Comparison of RVEN and node 213 signals. This figure shows two seismic waves recorded by sensor node 213 and a broadband seismometer located 56 m away. After time rectification, a 29 ms time shift produces an excellent match.

the data to our time rectification process. We call the difference between the true and apparent times the *timestamp error*. Figure 14 shows the results for nodes one and six hops away from the FTSP root. After rectification, 99.9% of the errors for the one-hop node and 93.1% of the errors for the six-hop node fall within our 10 ms error envelope.

7.6 Comparison with broadband station

Although time rectification works well in the laboratory, it is also necessary to evaluate its accuracy on the data collected during the field deployment. For this purpose, we made use of one of the broadband seismometer stations colocated with our sensor network. The RVEN (for "Reventador vent") station was located 56 m from sensor node 213. Given their proximity, we would expect the seismic waveforms captured by both RVEN and node 213 to be well correlated. Some time shift between the two signals would be expected: a seismic wave passing each station could be as slow as 1.5 km/sec, so the time lag between the signals could be as high as 37 ms. However, due to differences in the seismometers and the placement and ground coupling of the sensors, we would not expect perfectly correlated signals in every case.

We identified 28 events recorded by both RVEN and node 213. The data for node 213 was time rectified as described earlier, and the RVEN data was timestamped by the Reftek's internal GPS receiver. We applied a bandpass filter of 6–8 Hz to each signal to reduce sensor-specific artifacts. The cross-correlation between the signals produces a set of of *lag times* indicating possible time shifts between the two signals. Due to the periodic nature of the signals, this results in several lag times at multiples of the dominant signal period. For each lag time, we visually inspected how well the time-shifted signals overlapped and picked the best match by hand.

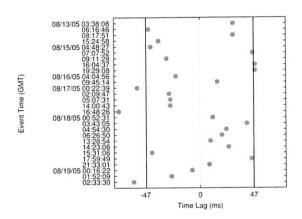


Figure 16: Lag times between Node 213 and RVEN. The best lag time between the two stations is shown for 28 events. best time lag between the two stations is shown. Most time shifts into the +/- 47 ms window that we would expect given the distance between the two stations and up to 10 ms of timing error.

Figure 15 shows an example of this process that demonstrates excellent correlation between the RVEN and node 213 signals with a 29 ms time shift. Figure 16 shows a scatterplot of the best lag times for all 28 events. Of these, only 5 events fall outside of a +/-47 ms window defined by the distance between the stations (+/-37 ms) and our acceptable sampling error (10 ms). We have high confidence that our time rectification process was able to recover accurate timing despite failures of the FTSP protocol.

8 Data Fidelity

The final and most important measure of our network is its ability to provide scientifically-meaningful data on the volcano's activity. In this section, we perform an initial analysis of the seismic and acoustic signals from a seismological perspective, with the goal of validating the accuracy of the signal quality and timing.

8.1 Acoustic wave propagation

The infrasonic (low-frequency acoustic) waves generated by the volcano are primarily the result of explosive events. We start by measuring the velocity of infrasonic waves recorded by our network, which is more straightforward than seismic analysis for several reasons. First, infrasonic waves generate a clear impulse in the microphone signal, making it easy to determine the time of the wave arrival at each sensor. In addition, acoustic waves propagate about an order of magnitude slower than seismic waves (roughly 340 m/s versus 1500-4000 m/s). We also expect an infrasonic wave to originate at the vent of the volcano, simplifying the wave velocity calculation.

We identified four events in our data set with a clear infrasonic component. For each event, we hand-picked

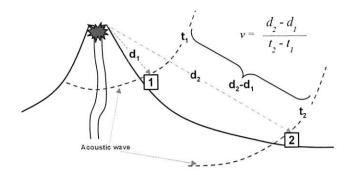


Figure 17: Computing acoustic wave velocity. The velocity of the acoustic wave is calculated based on the distance of each station from the vent, d_i , and the arrival time of the wave at each station, t_i .

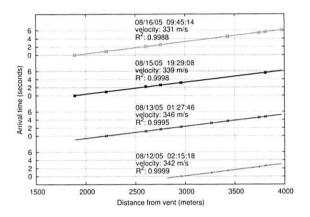


Figure 18: Acoustic wave arrival times and velocity. This figure shows the acoustic wave arrival time vs. distance from the vent for 4 separate events. Also shown is the resulting acoustic wave velocity and the \mathbb{R}^2 coefficient of determination.

the arrival time of the wave at each node using the timerectified signal. Figure 18 plots the wave arrival time versus the distance of each node from the vent. As shown in Figure 17, the velocity of the wave can be calculated by performing a linear regression on this dataset.

The result of this calculation is also shown in Figure 18. The velocity of sound in air is temperature-dependent; for temperatures between 10–20 °C the velocity range is 337–343 m/s. The calculated wave velocities are mostly in this range, with a mean of 339.5 m/s. The coefficients of determination R^2 are very high, between 0.9988 and 0.9999, showing that the timing and quality of the acoustic data closely matches our expectation of the infrasonic waves produced by the volcano.

8.2 Seismic wave propagation

Analyzing the seismic waves captured by our network is significantly more challenging. This is primarily because the source locations of seismic waves are unknown. Seismic events may originate at the vent (in the case of an ex-

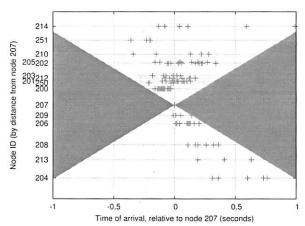


Figure 19: Time of arrival of each node over multiple events. This graph shows the spread of arrival times for each node. The arrival time and distance is relative to node 207. The arrival time for each node is fairly consistent over multiple events, with the exception of node 214. The shaded area indicates a move-out velocity of less than 1,500 m/s.

plosion) or deep within the edifice, producing very different patterns of P- and S-wave arrivals at each node. A full seismological analysis of our data is beyond the scope of this paper. However, we present a high-level measure of the *consistency* of the signals captured by our network: that is, we evaluate whether the seismic wave arrivals are consistent with expected volcanic activity.

The most natural measure of data consistency is whether the time of the seismic P-wave arrival at each sensor falls within an expected *envelope* based on the minimum speed at which seismic waves are believed to propagate at Reventador, which we estimate as 1500 m/s. We took 15 seismic events with clear P-wave arrivals and used an automatic algorithm [22] to determine the wave arrival time.²

Figure 19 shows a scatterplot of the arrival times with respect to node 207, which was chosen as an arbitrary reference point since data for this node appeared in all 15 events. The *y*-axis represents the distance of each node from 207. Depending on the seismic source location, we expect waves to arrive both before and after node 207. However, the slowest wave speed (1500 m/s) dictates the maximum difference in the wave arrival between each station.³ The shaded area in Figure 19 covers the "exclusion envelope" of arrival times at each station. As the figure shows, only 2 out of 124 arrivals fall outside of this envelope.

Finally, we take a closer look at two seismic events recorded by our array. Figures 20 and 21 show seismograms from each of the sensor nodes after time rectification. The *y*-axis corresponds to the distance of each node

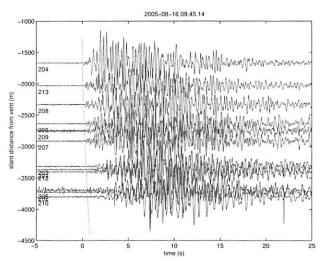


Figure 20: Explosion earthquake event at 08/16/2005 09:45:14 GMT. P-wave arrivals have been identified manually and a second-order polynomial (solid line) is fit to the arrivals. The arrival time move-outs are consistent with a shallow nearvent source.

from the vent. For each event, the P-wave arrivals have been determined by hand and a second-order polynomial has been fit to the arrival times at each node for clarity.

These two events show a very different pattern of wave arrival times. Figure 20 shows the seismic wave arriving first at stations near the vent (nodes 204 and 213). This is consistent with a shallow near-vent source corresponding to an explosion. This is confirmed by the corresponding acoustic data (shown in Figure 18) attributed to explosive expansion of gas.

In contrast, Figure 21 shows an event with the earliest arrivals in the middle of the sensor array and the endpoints relatively delayed; many such events were recorded by our network. This distribution implies a deeper source. At the same time, seismic velocity in the uppermost cone, which is comprised of unconsolidated volcanic deposits, is presumed to be slower. Such volcano-tectonic events are likely generated by the fracturing of solid media typically induced by pressurization within the edifice. This preliminary study demonstrates the value of our wireless sensor network for collecting accurate signals that can be subjected to seismological analysis.

9 Related Work

While the number of sensor network deployments described in the literature has been increasing, little prior work has focused on evaluating sensor networks from a scientific perspective. In addition, the high data rates and stringent timing accuracy requirements of volcano monitoring represent a departure from many of the previously-studied applications for sensor networks.

Low-data-rate monitoring: The first generation of sensor network deployments focused on distributed mon-

²Unlike acoustic waves, determining seismic wave arrival times is notoriously difficult. The seismograms in Figures 20 and Figure 21 should give the reader some appreciation for this.

³Note that there is no lower bound on arrival times, since a wave emanating from a deep source could arrive at all stations nearly simultaneously.

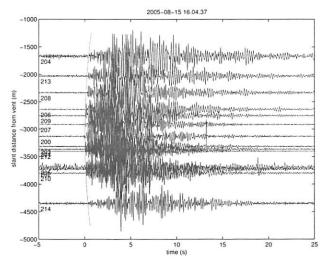


Figure 21: Tectonic earthquake event at 08/15/2005 16:04:37 GMT. In this event, seismic waves are first recorded near the middle of the sensor array. This is due either to a source closer to the center of the array, variations in velocity structure, or most likely both

itoring of environmental conditions. Representative projects include the Great Duck Island [24, 19, 10], Berkeley Redwood Forest [25], and James Reserve [2] deployments. These systems are characterized by low data rates (sampling intervals on the order of minutes) and very low-duty-cycle operation to conserve power. Research in this area has made valuable contributions in establishing sensor networks as a viable platform for scientific monitoring and developing essential components used in our work.

This previous work has not yet focused on the efficacy of a sensor network as a scientific instrument. The best example is the Berkeley Redwood Forest deployment [25], which involved 33 nodes monitoring the microclimate of a redwood tree for 44 days. Their study focuses on novel ways of visualizing and presenting the data captured by the sensor network, as well as on the data yield of the system. The authors show that the microclimactic measurements are consistent with existing models; however, no ground truth of the data is established. This paper highlights many of the challenges involved in using wireless sensors to augment or replace existing scientific instrumentation.

High-data-rate monitoring: A second class of sensor network applications involves relatively high data rates and precise timing of the captured signals. The two dominant applications in this area are structural health monitoring and condition-based maintenance. In each case, arrays of sensors are used to capture vibration or accelerometer waveforms that must be appropriately timestamped for later analysis.

NetSHM [15, 14, 30] is a wireless sensor network for structural health monitoring, which involves studying the response of buildings, bridges, and other structures to localize structural damage, e.g., following an earthquake. This system shares many of the challenges of geophysical monitoring; indeed, the data rates involved (500 Hz per channel) are higher than are typically used in volcano studies.

NetSHM implements reliable data collection using both hop-by-hop caching and end-to-end retransmissions. Their work explores the use of local computations on sensors to reduce bandwidth requirements. Rather than a global time-synchronization protocol, the base station timestamps each sample upon reception. The *residence time* of each sample as it flows from sensor to base is calculated based on measurements at each transmission hop and used to deduce the original sample time.

Several factors distinguish our work. First, NetSHM is designed to collect signals following controlled excitations of a structure, which simplifies scheduling. In our case, volcanic activity is bursty and highly variable, requiring more sophisticated approaches to event detection and data transfer. Second, NetSHM has been deployed in relatively dense networks, making data collection and time synchronization more robust. Third, to date the NetSHM evaluations have focused more on network performance and less on the fidelity of the extracted data. Other systems for wireless SHM include one developed by the Stanford Earthquake Engineering Center [9, 26] and earlier work by Berkeley on monitoring the Golden Gate Bridge [16].

Condition-based maintenance is another emerging area for wireless sensor networks. The typical approach is to collect vibration waveforms from equipment (e.g., chillers, pumps, etc.) and perform time- and frequency-domain analysis to determine when the equipment requires servicing. Intel Research has explored this area through two deployments at a fabrication plant and an oil tanker in the North Sea [8]. Although this application involves high sampling rates, it does not necessarily require time synchronization as signals from multiple sensors need not be correlated. The initial evaluation of these deployments only considers the network performance and does not address data fidelity issues.

10 Lessons Learned

Sensor network deployments, particularly in remote areas, involve significant cost in terms of time and equipment. Failures of hardware and software can have a negative impact on the uptake of this technology by domain science experts. Our experiences at Reventador have yielded a number of valuable lessons for future sensor network deployments.

1. Ground truth and self-validation mechanisms are critical: We did not initially consider colocating several of our wireless sensors with existing data loggers in order to establish ground truth. This would have clearly aided our analysis, though we were fortunate to locate one of our sensors near (but not immediately adjacent to) the RVEN sta-

tion. In addition, self-validation mechanisms are needed to provide detailed information on the health and accuracy of the data recorded by the network. The periodic "heartbeat" messages that we built into our system proved essential to remotely tracking system operation.

2. Coping with infrastructure and protocol failures: As discussed previously, the sensor nodes themselves were the most reliable components of the system. Even without classifying the 3-day network outage as an infrastructure failure, this downtime was far exceeded by outages caused by power failures at the base station. We did not devote enough attention to assuring the reliability of the base station and radio modem infrastructure, assuming it would be a trivial matter of plugging into wall power. This single point of failure was more fragile than expected.

Additionally, several pieces deployed software, including Deluge and FTSP, exhibited failures in the field than we not had expected given our laboratory experiments. These failures both speak for and show the limitations of careful, pre-deployment testing. We were fortunate to be able to correct protocol errors in the field and during post-processing, but the risk of uncorrectable problems will lead us towards more rigorous testing and analysis in the future.

3. Building confidence inside cross-domain scientific collaborations: It is important when working with domain scientists to understand their expectations and plan carefully to meet them. There is a clear tension between the desire of CS researchers to develop more interesting and sophisticated systems, and the needs of domain science, which relies upon thoroughly validated instrumentation. Pushing more complexity into the sensor network can improve lifetime and performance, but the resulting system must be carefully validated before deployment to ensure that the resulting data is scientifically accurate.

Good communication between CS and domain scientists is also critical. During the deployment, the seismologists were eager to see the collected signals, which were initially in an unprocessed format with timing errors as described earlier. From the CS perspective, the early data provided evidence of successful data collection, but from the geophysics perspective it highlighted failures in the time synchronization protocol. It took a great deal of effort after the deployment to build confidence in the validity of our data.

11 Conclusions and Future Work

As sensor networks continue to evolve for scientific monitoring, taking a domain science-centric view of their capabilities is essential. In this paper, we have attempted to understand how well a wireless sensor network can serve as a scientific instrument for volcano monitoring. We have presented an evaluation of the data fidelity and yield of a real sensor network deployment, subjecting the system to the rigorous standards expected for geophysical instrumen-

tation.

We find that wireless sensors have great potential for rapid and dense instrumentation of active volcanoes, although challenges remain including improving reliability and validating the timing accuracy of captured signals. The network was able to detect and retrieve data for a large number of seismic events, although our event detection parameters require tuning to capture more signals. In terms of reliability, base station outages affected the network about 27% of the time during our deployment, with a single software failure causing a 3-day outage. However, nodes appeared to exhibit an uptime of 96%, which is very encouraging. Clearly, work is needed to improve the robustness of the base station and system software infrastructure.

Our most difficult challenge was correcting the timing in the captured signals and validating timing accuracy. A comparative analysis against a GPS-synchronized standalone data logger shows very good correlation: 23 out of 28 events correlated against the Reftek broadband station exhibited lag times within the expected 47 ms window. Across the sensor array, only 2 out of 124 P-wave arrivals fall outside of an expected velocity envelope, suggesting that our timing rectification is very consistent. This is further reinforced by linear regression of acoustic wave arrival times with R^2 values of greater than 0.99. Finally, preliminary analysis of the recorded seismic signals is consistent with expected volcanic activity, exhibiting differences between explosion-related activity and deep-source events.

Future directions: Our group is continuing to develop sensor networks for volcano monitoring and we expect to conduct future deployments. Our eventual goal is to design a large (50 to 100 node) sensor array capable of operating autonomously for an entire field season of three months.

A primary concern for future work is reducing power consumption to extend network lifetimes. Although we did not experience power-related failures of sensor nodes, we were fortunate that the deployment logistics permitted us to change batteries as needed. The highest power draw on our platform is the sampling board, which cannot be powered down since we must sample continuously. One approach is to perform more extensive signal analysis on the sensor nodes to reduce the amount of data that must be transmitted following an event. However, geophysicists are accustomed to obtaining complete signals, so we must balance network lifetime with signal fidelity.

In addition, we are interested in exploring novel approaches to programming large sensor arrays to perform collaborative signal processing. Domain scientists should not have to concern themselves with the details of sensor node programming. We plan to develop a high-level programming interface to facilitate more rapid adoption of this technology.

Acknowledgments

The authors wish to thank Thaddeus Fulford-Jones and Jim MacArthur for their assistance with hardware design; Omar Marcillo and Mario Ruiz for assistance with the field deployment; and the staff of the Instituto Geofísico, IGEPN, Ecuador, for logistical and hardware development support. Finally, many thanks to our shepherd, Chandu Thekkath, whose suggestions greatly improved the paper. This project is supported by the National Science Foundation under grant numbers CNS-0519675 and CNS-0531631.

References

- R. Cardell-Oliver. Rope: A reactive, opportunistic protocol for environment monitoring sensor network. In Proc. The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II), May 2005.
- [2] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In *Proc. the Workshop on Data Communications in Latin America and the Caribbean*, Apr. 2001.
- [3] Crossbow Technology Inc. http://www.xbow.com.
- [4] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In Fifth Symposium on Operating Systems Design and Implementation, December 2002.
- [5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In Proc. the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 93–104, Boston, MA, USA, Nov. 2000.
- [6] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, November 2004.
- [7] J. Johnson, R. Aster, and P. Kyle. Volcanic eruptions observed with infrasound. *Geophys. Res. Lett.*, 31(L14604):doi:10.1029/2004GL020020, 2004.
- [8] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea. In SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems, pages 64–75, New York, NY, USA, 2005. ACM Press.
- [9] J. P. Lynch, Y. Wang, K.-C. Lu, T.-C. Hou, and C.-H. Loh. Postseismic damage assessment of steel structures instrumented with self-interrogating wireless sensors. In *Proceedings of the 8th Na*tional Conference on Earthquake Engineering, 2006.
- [10] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02), Atlanta, GA, USA, Sept. 2002.
- [11] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In Second ACM Conference on Embedded Networked Sensor Systems, November 2004.
- [12] S. McNutt. Seismic monitoring and eruption forecasting of volcanoes: A review of the state of the art and case histories. In Scarpa and Tilling, editors, *Monitoring and Mitigation of Volcano Hazards*, pages 99–146. Springer-Verlag Berlin Heidelberg, 1996.
- [13] Moteiv, Inc. http://www.moteiv.com.
- [14] J. Paek, K. Chintalapudi, J. Caffrey, R. Govindan, and S. Masri. A wireless sensor network for structural health monitoring: Performance and experience. In *Proc. The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, May 2005.
- [15] J. Paek, N. Kothari, K. Chintalapudi, S. Rangwala, N. Xu, J. Caffrey, R. Govindan, S. Masri, J. Wallace, and D. Whang. The performance of a wireless sensor network for structural health monitoring.

- [16] S. N. Pakzad, S. Kim, G. L. Fenves, S. D. Glaser, D. E. Culler, and J. W. Demmel. Multi-purpose wireless accelerometers for civil infrastructure monitoring. In *Proc. 5th International Workshop on Structural Health Monitoring (IWSHM 2005)*, Stanford, CA, September 2005.
- [17] V. Paxson. On calibrating measurements of packet transit times. In Measurement and Modeling of Computer Systems, pages 11–21, 1998.
- [18] V. Paxson. Strategies for sound internet measurement. In IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement, pages 263–271, New York, NY, USA, 2004. ACM Press
- [19] J. Polastre. Design and implementation of wireless sensor networks for habitat monitoring. Master's thesis, University of California at Berkeley, 2003.
- [20] Refraction Technology Inc. http://www.reftek.com.
- [21] R. Scarpa and R. Tilling. Monitoring and Mitigation of Volcano Hazards. Springer-Verlag, Berlin, 1996.
- [22] R. Sleeman and T. van Eck. Robust automatic p-phase picking: an on-line implementation in the analysis of broadband seismogram recordings. Phys. Earth Planet. Int, 1999, 113, 1-4, 265-275.
- [23] R. Szewczyk, A. Mainwaring, J. Polastre, and D. Culler. An analysis of a large scale habitat monitoring application. In *Proc. Second ACM Conference on Embedded Networked Sensor Systems (Sen-Sys)*, 2004.
- [24] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proceedings of the First Euro*pean Workshop on Sensor Networks (EWSN), January 2004.
- [25] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *Proc. the Third ACM Conference* on Embedded Networked Sensor Systems (SenSys 2005), November 2005.
- [26] Y. Wang, J. P. Lynch, and K. H. Law. A wireless structural health monitoring system with multithreaded sensing devices: Design and validation. In *Structure and Infrastructure Engineering*, 2005.
- [27] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In Proc. Second European Workshop on Wireless Sensor Networks (EWSN'05), January 2005.
- [28] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh. Deploying a wireless sensor network on an active volcano. *IEEE Internet Computing, Special Issue on Data-Driven Applications in Sensor Networks*, March/April 2006.
- [29] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003), November 2003.
- [30] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proc. ACM SenSys '04*, November 2004.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- · fostering technical excellence and innovation
- · encouraging computing outreach in the community at large
- · providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to ;login:, the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- · The right to vote in USENIX Association elections
- · Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- · Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications: see http://www.usenix.org/membership/specialdisc.html

For more information about membership, conferences, or publications, see http://www.usenix.org.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- · Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- · Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at http://www.sage.org.

Thanks to USENIX & SAGE Supporting Members

Addison-Wesley Professional/ Prentice Hall Professional

Ajava Systems, Inc.

AMD

Cambridge Computer Services, Inc.

cPacket Networks

DigiCert® SSL Certification

EAGLE Software, Inc.

Electronic Frontier Foundation

FOTO SEARCH Stock Footage and Stock Photography

GroundWork Open Source

Solutions

Hewlett-Packard

IBM

Infosys Intel

Interhack

Microsoft Research
MSB Associates

NetApp Oracle OSDL

Raytheon Ripe NCC

Sendmail, Inc.

Splunk

Sun Microsystems, Inc.

Taos

Tellme Networks

UUNET Technologies, Inc.

VMware

